ARCHITECTURAL AND CIRCUIT ISSUES FOR A HIGH CLOCK RATE FLOATING-POINT PROCESSOR

TECHNICAL REPORT NO. SSEL-251

1995

by

Thomas Richard Huff

SOLID-STATE
ELECTRONICS
LABORATORY

DEPARTMENT OF ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE
THE UNIVERSITY OF MICHIGAN, ANN ARBOR

SF 298 MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	~~March 20, 1996~~	Technical Report

4. TITLE AND SUBTITLE

Architectural and
Circuit Issues for a High Clock Rate Floating-Point Processor

5. FUNDING NUMBERS

DAAH04-94-G-0327

6. AUTHOR(S)

Thomas Richard Huff

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)

University of Michigan
Dept. of Electrical Engineering & Computer Science
1301 Beal Ave.
Ann Arbor, MI 48109-2122

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

ARO 33790.19-EL

11. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*.

This work examines the issues confronting the designer of floating-point units for high-performance microprocessors. Sophisticated hardware coprocessors for floating-point arithmetic have been pursued primarily within the past decade. The development of these coprocessors parallels that of integer processors; initially simple designs were altered to satisfy the demand for increased performance. Architectural optimizations and technology improvements have had the greatest effect on performance. This work will examine these issues specifically by determining the mechanisms through which a floating-point unit can stall instruction execution, and by describing the implementation and verification of a GaAs floating-point design. This work represents a unique, comprehensive, and accessible study of important issues for supporting high-performance floating-point execution.

The culmination of this work has been the design of an IEEE-754 compliant double precision floating-point unit; the chip was designed in a 1.0 um GaAs direct-coupled FET logic process. Most of the conclusions regarding architectural optimizations are independent of technology, though a number of trade-offs in the design were made within the constraints of integration levels, fanin, fanout, logic topologies, speed, and power of GaAs direct-coupled FET logic. The final FPU achieves a high level of performance that exceeds many current leading commercial processors.

14. SUBJECT TERMS

15. NUMBER IF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

This report has also been submitted as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the University of Michigan, 1995.

This work was supported in part by the Advanced Research Projects Agency under Grant DAAH04-94-G-0327.

ABSTRACT

Architectural and Circuit Issues for a High Clock Rate Floating-Point Processor

by

Thomas Richard Huff

Chair: Professor Richard B. Brown

This dissertation examines the issues confronting the designer of floating-point units for high-performance microprocessors. Sophisticated hardware coprocessors for floating-point arithmetic have been pursued primarily within the past decade. The development of these coprocessors parallels that of integer processors; initially simple designs were altered to satisfy the demand for increased performance. Architectural optimizations and technology improvements have had the greatest effect on performance. This work will examine these issues specifically by determining the mechanisms through which a floating-point unit can stall instruction execution, and by describing the implementation and verification of a GaAs floating-point design. This dissertation represents a unique, comprehensive, and accessible study of important issues for supporting high-performance floating-point execution.

A synchronization problem exists between the integer and floating-point units that causes the FPU to stall the IPU. This can be overcome through the use of decoupling data

and instruction queues, a reorder buffer, and result busses. Increasing the number of queue or reorder buffer entries results in improved performance that cannot be equalled either through pipelining the FPU functional units, or by attempts to reduce floating-point functional unit latency, both of which require a significant increase in resources.

One important class of stall conditions can be addressed by: analyzing memory system characteristics; code scheduling to improve FPU performance on commonly encountered instruction sequences; selection of the FPU instruction and data transfer point in the integer pipeline; and the degree of instruction issue. Instruction issue policies attempt to exploit available parallelism that exists in the instruction stream. Different policies offer design points which, while achieving similar performance, vary with respect to design complexity and resource requirements. The most promising designs emphasize either the extraction of instruction-level parallelism through greater complexity, or focus on simplicity to increase clock frequency. Verification consumes an ever-increasing share of design time as processors become more complex. Methods of functional and performance validation of the FPU are discussed. Several utilities were created to support implementation of the high-speed VLSI chips used in the project, and suggestions for an automated approach to performing timing analysis and logic optimization are presented.

The culmination of this work has been the design of an IEEE-754 compliant double precision floating-point unit; the chip was designed in a 1.0μm GaAs direct-coupled FET logic process. Most of the conclusions regarding architectural optimizations are independent of technology, though a number of trade-offs in the design were made within the constraints of integration levels, fanin, fanout, logic topologies, speed, and power of GaAs direct-coupled FET logic. The final FPU achieves a high level of performance that exceeds many current leading commercial processors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1
Introduction

The use of sophisticated hardware coprocessors for floating-point computations has occurred primarily within the past 8 to 10 years. VLSI chips devoted to floating-point arithmetic appeared in the early 1980's and at first offered only single-chip adders and multipliers. These tended not to be pipelined and often required external control units and register sets. The appearance of high performance workstations has resulted in an increasing number of applications that utilize floating-point arithmetic. Fields such as computer graphics, which in the past have depended on integer arithmetic, are moving toward specialized floating-point graphics processors. In addition, the past decade has seen significant growth in digital signal processing, and a similar move away from range and precision constraints through the use of floating-point numbers. In many ways, the brief history of floating-point processors parallels that of integer processors. Designs were initially simple, but with time, performance gains were achieved through both architectural optimizations and technology improvements which led to increases in complexity. Processor performance has improved at a uniform rate of 50% per year over the past 10 years [Upton94]; Figure 1.1 shows that floating-point unit clock frequency has also increased approximately 50% per year (refer to Appendix C for the references). Much of the increase in clock frequency can be attributed to better process technology. As Figure 1.2 shows, there is also a corresponding increase in the amount of circuitry used in floating-point units. In particular, addition and multiplication algorithms have benefitted from optimizations which have reduced the latency of both to as few as 2 cycles. Addition algorithms have been optimized through leading zero prediction and the mutually exclusive characteristics of normalization, alignment, and rounding. Multiplication improvements have been due to the use of Wallace arrays and Booth recoding.

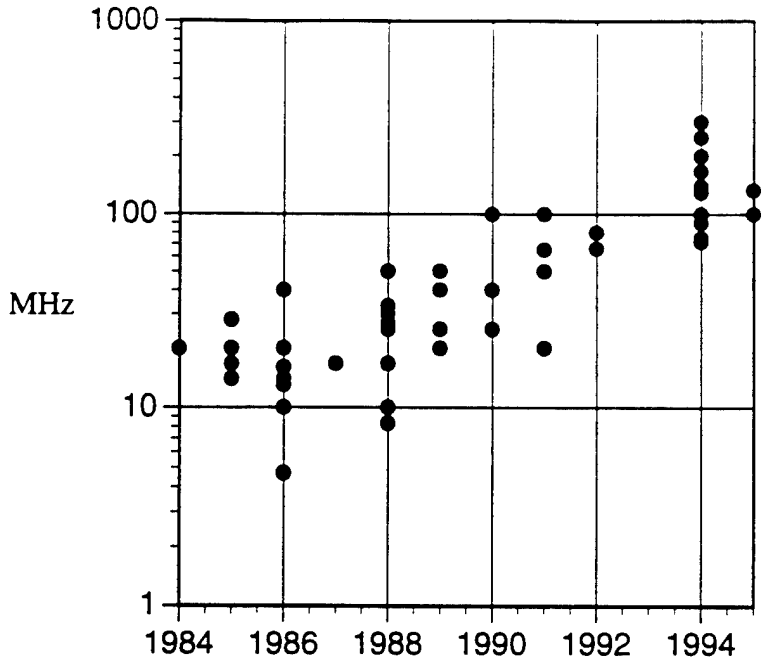


Figure 1.1 FPU Clock Frequency vs Year

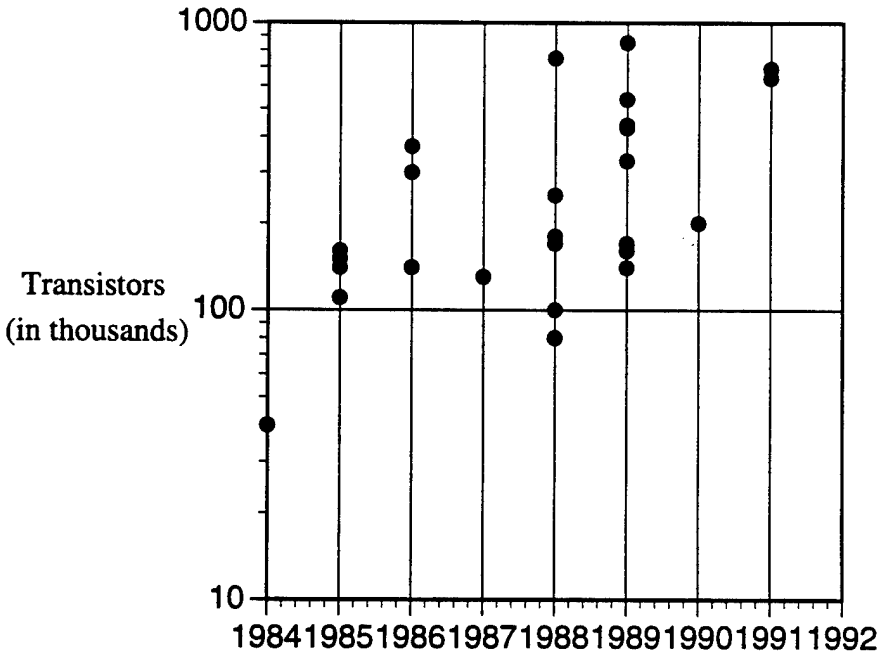


Figure 1.2 FPU Transistor Count vs Year

Integer processing units (IPU's) and floating-point units (FPU's) share a similar history of applying more complex architectural solutions in order to gain performance. In the early 1980's, microprocessors were not pipelined, did not contain on-chip caches, and often required many cycles to complete an instruction. In recent years many supercomputer

features have been applied to the design of microprocessors, including pipelining, caches, load/store instruction sets, multiple execution units, higher degrees of instruction issue, and virtual memory. While some of these approaches have been applied to floating-point design, a more complete investigation of the design space is warranted, with the goal being a latency tolerant high-performance GaAs FPU. This dissertation represents a unique, comprehensive, and accessible study of important issues for supporting high-performance floating-point execution.

The development of a processor system involves four steps: evaluation of micro-architectural choices in order to minimize cycles-per-instruction (CPI), implementation of a circuit that efficiently supports the architecture in light of technology constraints, functional validation, and critical path analysis to optimize clock frequency. Architectural experiments focus on identifying bottlenecks that limit performance, such as conditions that generate stalls. A number of metrics can be used in the architectural studies. The progress of individual instruction types through the machine can be tracked by the average latency that an instruction takes to reach the various points of interest: the floating-point instruction staging area, the issue point within the FPU, the completion of the instruction into the reorder buffer, and the write-back of the register file. Additional parameters that are important to consider include dynamic instruction frequencies, basic block size, bus utilization, average degree of issue, sizes for different types of resources, and of course CPI. These measurable quantities will be used to discover ways of improving performance.

Processor performance has been progressing at a historical rate of about one percent per week, and the design and verification time for any additional feature should be justified against this standard. An improvement of less than 10% is of doubtful benefit, unless it is very simple to implement, especially in consideration of the accuracy limitations of most analysis techniques. For example, while trace-driven simulation provides access to billions of instructions, the design space is so broad and the possible experiments are so varied that simulating this amount of instructions may not be reasonable for each run. Inevitably, questions arise about how to improve the simulation speed and how to ensure the veracity of the results. These issues will be briefly examined in the context of instruction sampling and er-

ror sources.

The architectural experiments presented in this dissertation begin with the analysis of three issue policies for floating-point instructions. These policies specify whether instructions issue and complete in order or out of order; each requires a different degree of resources and design complexity. Integration levels are an order of magnitude lower for GaAs than for CMOS; this drives many decisions concerning resource allocation. The next architectural experiment sought to answer the question: how sensitive is overall performance to the latency of floating-point functional units? Approaches for reducing latency tend to require an increase in resources, since more conditions need to be resolved earlier in time through the use of parallel logic. The effects of pipelining functional units impacts chip area. A designer needs to know how the cost/benefit ratio compares to that of adding other features, such as additional reorder buffer entries. Design time must also be considered, since lower-latency functional units are often more complex and require more validation. Other questions related to the characteristics of the functional units include: what applications use square root and does it make sense to support this operation in hardware; are addition operations common enough to warrant a second add unit; can division be performed within the multiplication unit without degrading the performance of multiply instructions; what precision operand formats should be supported since these also will have an impact on area and critical paths?

Since integration levels are low for GaAs, the IPU and FPU need to be partitioned into separate chips, which increases the latency of floating-point operations, due to chip crossings. Queues can be used to decouple these units and allow more instruction slip, but this approach has an impact on the support of precise exceptions. The dissertation discusses different characteristics of memory and floating-point computation exceptions, and shows how performance will be affected by each type. There may also be an intrinsic difference in clock frequencies between the IPU and FPU, which queues can hide.

Memory access time has improved at a much slower rate than processor speed; supplying instructions and data for a machine with a 4ns clock period becomes difficult. In GaAs, integration levels severely constrain the amount of cache that can reside on-chip, so

other techniques are needed to offset the corresponding loss in performance, such as prefetching and higher bandwidth for memory accesses.

In currently-available packaging technology, a multiple chip design places demands on the limited number of package pins. This argues for sharing I/O pins between the FPU and data cache. The impact of this approach is evaluated. These interfacing issues also affect the point in the integer pipeline at which floating-point instructions are transferred to the FPU. Depending on the degree of issue and the latency of the first-level data cache, a later transfer point might add unnecessary latency to every floating-point instruction. Balance between processing and communication resources is also important as instruction parallelism increases within the FPU.

In the multidimensional design space briefly described above, different design points may achieve similar performance in very different ways. For instance, a complex design might implement dual issue and an out-of-order completion policy using a reorder buffer, in effect trading clock frequency for a decrease in CPI. On the other hand, a simpler design might forgo the use of a reorder buffer, choosing instead an in-order issue and completion policy and a more conservative mechanism for supporting precise memory exceptions. If the performance of these two implementations is similar, the simple design would be favored for the benefit of its shorter design cycle and an ability to more easily track technology improvements.

Once a micro-architecture has been defined, a specific implementation is chosen that is suitable to the features and limitations of the target technology. For example, several types of adders are used in various parts of the FPU design; issues such as fanin, fanout, logic topologies, area, speed, and power determine which designs are best suited for a GaAs processor. Most of the functional unit designs used in the FPU originated with work done elsewhere. These schemes, all of which strive to reduce latency, are evaluated in this dissertation in the context of GaAs technology. Several issues which have been overlooked in those designs will be briefly discussed and a novel approach for the design of a conversion unit will be presented. At a higher level, the dissertation will examine predecoding to reduce the gate-depth of critical issue logic and ways of balancing observability during test-

ing versus design time, resource requirements, and impact on critical paths. Methods of supporting floating-point loads, stores, and precise memory exceptions will also be discussed. Finally, functional verification of a complex design will be examined primarily in the context of random testing, both at the functional unit level and at the higher instruction-stream level.

The last component of a processor design involves the analysis tools that provide feedback about timing and functionality. Often, custom utilities need to be written in order to address a specific need, and these point tools must be developed quickly since their absence can delay a particular phase of the design. The programming environment chosen for the tool will depend on the nature of the problem, including both how large a design the utility will be used for and how often this particular analysis will be performed. Chapter 5 will discuss a number of tools that have been created in order to enable different analysis capabilities, beginning with a delay calculator that supports accurate determination of GaAs circuit delays. This utility which provides input to static timing analysis is essentially a recursive network traversal engine, which in turn is the core of several other tools. For example, a set of tools based on this engine has been developed to support designs such as Aurora III, which uses a two-phase clocking scheme. In spite of a designer's best intentions, identifying all hazards is difficult without an automated approach; missing even one such error can seriously reduce the effective clock frequency. Control of clock skew is also an important issue for high-frequency designs, and a second set of utilities built on the network traversal engine has been created to extract the clock distribution network and generate a ready-to-run HSPICE netlist. Information derived from simulating this netlist is used in several ways, including identification of latch-to-latch skew, clock transit time to any point on a chip, and wire sizing along the distribution network to control interconnect resistance. Timing analysis also focuses on reducing gate-depth along critical paths, and a third utility supports this type of analysis. A latch-based design allows time to be borrowed from the phase that precedes or follows a critical path. Consequently, the target logic depth of 20 gates per phase can be relaxed in certain instances if the worst-case previous path is shorter than this threshold. The levelizer utility generates 2D and 3D histograms that enable

the designer to quickly identify sections of logic that require improvement.

Finally, an automated high level timing methodology is described as a motivation for future work. During the design of the FPU, it became apparent that a large class of timing optimizations currently must be performed by the user. These transformations tend to be quite mechanical and include factoring out late-arriving signals, pattern matching logic optimization, manual retiming, and buffer selection and sizing. Altogether, these different actions could be collected into an extremely effective automated timing analysis and optimization system.

The observations of this dissertation have culminated in the implementation of a GaAs FPU which achieves a high level of performance comparable to current leading commercial processors. Based on the extensive simulations of this study, this FPU delivers this performance while only infrequently stalling the integer processing unit. The design is IEEE compliant with respect to rounding modes and exceptions, supports 40 floating-point instructions, consists of 500,000 transistors, operates at a clock frequency of 250MHz, and achieves a SPECfp92 rating of greater than 300.

CHAPTER 2
Circuit Issues for GaAs

2.1 Description of the Technology

Direct-coupled FET Logic (DCFL) gates are similar in topology to NMOS, with inverters and NOR gates comprising the basic building blocks. Enhancement pulldown and depletion pullup devices are ratioed in such a way as to provide desired output high and low voltages over normal operating conditions. The depletion device is source-gate connected to provide a current source. Gate delay for an unloaded device is on the order of 60ps and loaded gates typically have delays in the range of 100ps to 150ps. The gate of a MESFET is actually a Schottky diode; there is no gate dielectric as is in MOSFET devices. This diode gate introduces several unique issues into the design of VLSI circuits, one example being the small voltage swing for these devices. Since the gate is a diode, the gate voltage for a logic high is clamped to a single diode drop, on the order of 0.6 volts. For logic gates to function properly with such low output-high voltages, the enhancement transistors must have small threshold voltages, typically about 0.2 volts. Consequently, designs are sensitive to voltage drops along the ground rail. A top-level aluminum interconnect plane is used to provide a clean ground with less than 20mV of noise; each cell connects directly to this plane. IR drops along Vdd are not as critical and gates operate correctly with little loss in speed for a power supply voltage as low as 1.2 volts. Power is routed in Metal 3 and is sized such that no gate sees an IR drop along Vdd of more than 0.5 volts.

The diode gate of a MESFET also results in an unusual transfer characteristic, as shown for the NOR gate of Figure 2.1. As the input voltage increases beyond 0.8 volts, the output voltage begins to rise; above a certain input voltage the output erroneously becomes a logic one. When this phenomenon occurs, the diode gate current is large and the gate-

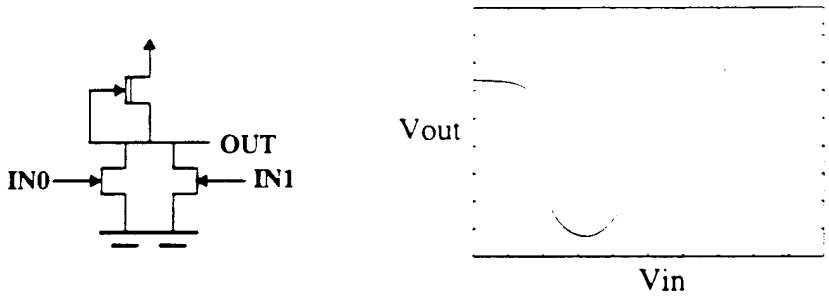


Figure 2.1 Two-input NOR and Transfer Function

drain junction becomes forward biased. Though DCFL signals are normally clamped by the driven gates, when large buffers are used to quickly change the state of highly capacitive interconnect the overdriving condition may occur. The buffer used to drive such a net may provide a current which is appropriate for charging the wire but is too large for destination gates. A solution used for the Aurora I design (the first of three GaAs microprocessors designed in our research group) involved placing a diode at the output of each buffer cell in order to clamp the output high signal to one threshold voltage. A more effective feedback approach for buffering was subsequently obtained from Vitesse and has been used for later designs. Shown in Figure 2.2 [Fulkerson91], this buffer provides a large transient current to charge a wire, then reduces its drive, providing a smaller current to maintain a stable logic high voltage. The small feedback transistor (typically 5 microns wide) serves the same purpose as the diode used in the earlier design, but has the advantage of more quickly discharging the internal node of the buffer. To improve the noise margins and yield of this construct, a small diode is often added at the drain of the feedback transistor. This diode serves to boost the voltage level of the internal node, and consequently the level of a logic high. The feedback transistor is on only while driving a logic high. For a logic low, the pullup transistor of the output stage is off and the pulldown transistor acts as a current sink. This

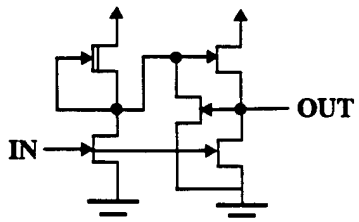


Figure 2.2 Feedback Buffer

behavior adds a frequency-dependent component to overall power dissipation, whereas the power dissipation of conventional DCFL gates is almost independent of the operating speed. These buffers are among the most commonly used cells in a design, and can contribute more than 40% of the power of a chip; to be accurate, estimation of power dissipation must reflect this dynamic power component.

Another characteristic of the GaAs technology being discussed is a high transistor source resistance, which tends to limit the use of stacked transistor logic, such as NAND gates. A 2-input NAND represents the highest degree of stacking that is allowable, but does not offer a speed advantage compared to an inverter-NOR version of the same function. The use of only DCFL NOR gates can increase the number of levels along critical paths unless special circuit structures are used; critical path lengths for the Aurora II design had 15% more gates than a comparable CMOS implementation. One such structure used extensively for later designs is an Earle latch that combines a 2-input mux with a latch and a high-current output buffer. The latch/output-buffer operates in a feedback mode similar to that just discussed. This circuit accounted for 40% of the circuit area of the Aurora II design.

Leakage currents, which are several orders of magnitude larger in GaAs than silicon devices, constrain the maximum fanin of a DCFL gate to four inputs. More inputs reduce the logic high level, especially at higher temperatures. However, there are times when a larger fanin gate is needed to reduce delay along a critical path. A solution can be found by extending the feedback buffer to support the additional inputs.

Because this gate type ensures that output pullup and pulldown transistors are not on simultaneously, one can size these transistors solely for their driving capability and not in consideration of noise margins. As a result, the pullup device will be large enough to accommodate the leakage currents of a reasonable number of pulldown devices, while still providing a sound logic one. Figure 2.3 shows a 4 input version of this type of gate. These buffered gates need to be used judiciously since they are more costly in terms of area than their DCFL counterparts, as shown in Table 2.1. The 5- to 8-input DCFL areas listed in this table are estimates, since these gates are not functionally reliable across process and temperature corners and thus have not been implemented. The buffered versions of these gates

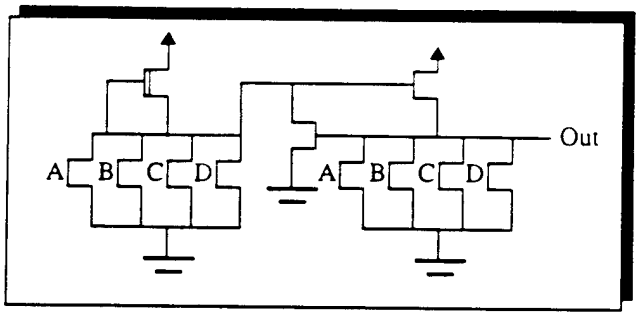


Figure 2.3 4-Input NOR Squeeze Gate

use large transistors for the output stage; the difference in area could be reduced by using smaller transistors, but at the expense of capacitive driving capability.

The diode gate also makes pass gates and dynamic logic more difficult or altogether impractical. The control and data signals to pass gates need to be driven by buffered cells to operate correctly at high temperatures. The register file design in the first two Aurora chips made use of a pass gate latch in order to reduce area, and was simulated thoroughly in order to ensure functionality. Dynamic logic tends to be difficult to implement in GaAs due to the current which flows into the gate terminal of MESFET transistors. This current is typically small, on the order of 10 to 30 uA, but can still be disruptive for dynamic pre-charging of nodes. Accurate analysis tools are a necessity when utilizing dynamic logic and a failure in this area can result in a design which is not functional at any frequency. Current

Table 2.1 Area Comparison of DCFL and Buffered NOR Gates

Fanin	DCFL Area (um2) *estimated	Buffered Area (um2)	% Difference
1	558	1364	144
2	823	1730	110
3	1101	2021	84
4	1401	2294	63
5	1730*	2591	50
6	2095*	2876	37
7	2503*	3162	26
8	2961*	3434	16

into the gate of transistors also places a constraint on how much fanout is tolerable, since the loads connected to a driving gate may exceed the current sourcing capability of that gate. The distribution of reset is a case where performance may not be an issue and fanout may be quite large. Analysis tools need to identify situations like this in order to ensure proper functionality.

2.2 Importance of Interconnect

The importance of interconnect in a VLSI process cannot be overstated. The switching delay, τ, for any logic family, is related to the difference in charge between states at the output of a logic gate, and to the current available to effect a change of state: $\tau \propto \frac{C\Delta V}{I}$. Sensitivity to parasitic loading varies with process and logic family. In FET technologies, this is the dominant delay mechanism; it calls for small logic swings, high transconductance, and low parasitic capacitance.

Most of the parasitic capacitance comes from interconnect. Of primary importance is keeping the circuit area as small as possible to minimize wire length; this reduces both parasitic capacitance and time-of-flight for signals. Routing capacitance is minimized by using sufficient levels of interconnect, narrower lines, larger separation between interconnect layers, and lower dielectric-constant insulators. The effect of reducing the space between lines, as is done when processes are scaled, is not immediately obvious; while it reduces the circuit area, it does increase horizontal line-to-line capacitance. To evaluate the significance of interconnect on overall area utilization, we looked at an 8x8 multiplier implemented in several GaAs process technologies. The total-routing-area data shown in Table 2.2 makes a strong case for reducing interconnect spacing to the fabrication limit [Brown92b]. Table 2.2 also demonstrates how smaller transistor dimensions have a smaller effect on overall layout utilization.

The importance of minimizing interconnect capacitance is illustrated in Figure 2.4 and Figure 2.5, which show the effects of reducing capacitive load and of reducing unloaded gate delay on four critical paths in the Aurora II microprocessor. The logic paths in these

Table 2.2 Comparison of 8x8 Multipliers in Three DCFL Processes

	Gate Metal	Metal 1	Metal 2	Metal 3	Total Layout Area	Total Routing Area
Process A	1.00	1.00	1.00	1.00	1.00	1.00
Process B	0.90	0.60	0.50	0.28	0.49	0.21
Process C	0.50	0.97	1.11	1.43	0.97	0.82

plots are from the register file (RF), adder (A1 and A2), and branch logic (BR). (These figures ignore the fact that faster gates would have greater transconductance and therefore drive the capacitive loads more effectively). The plots show clearly that reducing interconnect capacitance would be even more effective at increasing circuit speed than would reducing intrinsic gate delay. The effects on path delay vary among the paths simulated. The smallest difference in results is for the branch logic, where a 50% reduction in capacitance has a 40% greater effect than a similar reduction in unloaded gate delay. The biggest difference is in the register file, where capacitance has a 248% greater effect. The branch path consists of a large number of lightly loaded paths, whereas the RF path involves a smaller number of heavily loaded gates.

The importance of having enough layers of interconnect merits further illustration. In our designs, Gate Metal and Metal 1 are used for wiring inside of leaf cells, and Metals

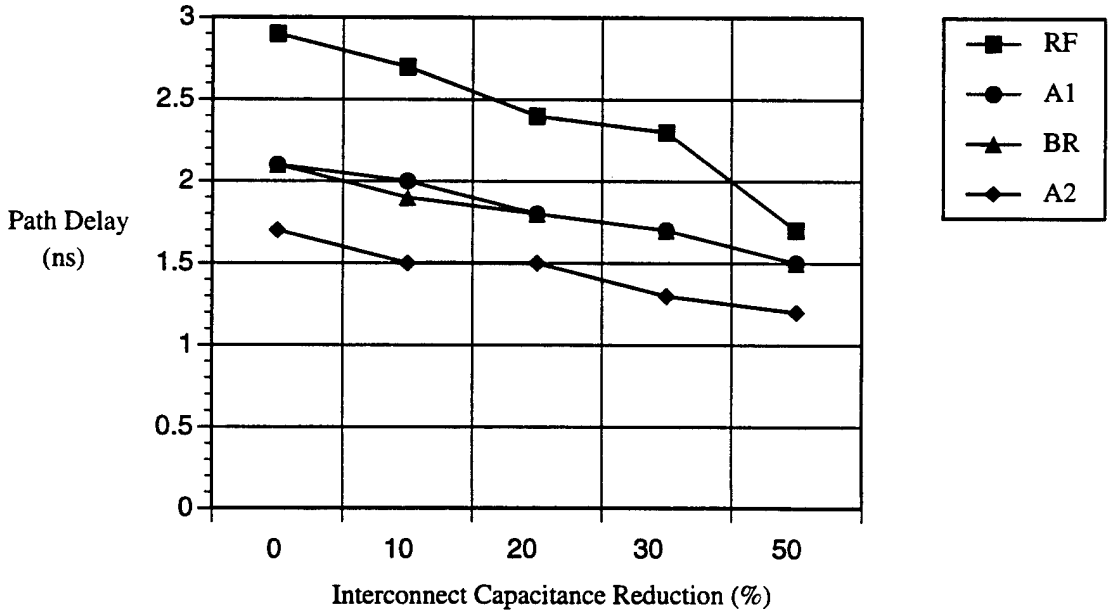


Figure 2.4 Interconnect Capacitance Reduction (%)

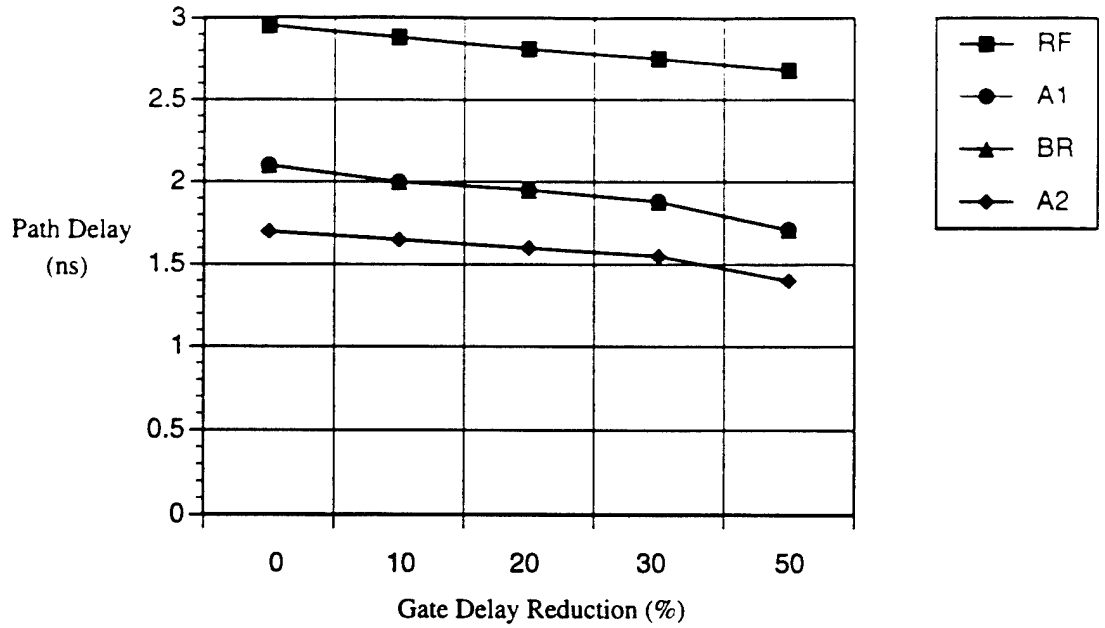


Figure 2.5 Unloaded Gate Delay Reduction (%)

1, 2, and 3 are used for datapath, standard cell, and global routing. Metal 4 is a ground plane, and Vdd is distributed on Metal 3. Table 2.3 shows the improvement in density which resulted in moving from HGaAs II (a 3-metal process) to HGaAs III (a 4-metal process). Of course, geometric design rule changes between the processes and other factors also effect the density, but cells tend to be interconnect-limited instead of device-limited. The control blocks are different circuits (bypass logic in HGaAs II and stall logic in HGaAs III), but they are about the same size, and both are implemented in standard cells using the same logic synthesis tool (Finesse, from Cascade Design Automation). The register files in Table 2.3 are both 32-word x 32-bit, three-port, tree-decoded, pass-gate latch implementations, which differ only in buffering.

The density numbers for both CPU's include all of the unoccupied space in the pad frame - there is actually more unused space in the version with 4-metal interconnect. Some of the increase in density is due to the inclusion of additional memory structures for the small on-chip instruction cache on the 4-metal chip. But even when this difference is factored out, the HGaAs III version of the CPU is still about 2.4 times denser. In this analysis, half of the improvement is due to the third layer of routing; improved circuit structures and layout techniques incorporated into newer CAD tools account for another 35%, and the re-

Table 2.3 Density comparison between 3-metal and 4-metal processes.

Circuit	HGaAs II Transistor Count	Density (Trans./mm^2)	HGaAs III Transistor Count	Density (Trans./mm^2)
Largest Control Block	582	1067	516	1364
Register File	21,910	2014	23,278	4253
CPU	60,500	540	160,000	1475

maining 15% of improvement results from smaller line widths in the HGaAs III process.

Adding interconnect layers to a digital process beyond a routeable gate metal, 3 interconnect levels, and a ground plane would result in diminishing returns. Trying to achieve high performance in a DCFL process with fewer than five layers or with a coarse interconnect pitch or an inefficient design style, though, starts a vicious cycle. A larger layout has more capacitance, therefore requiring larger buffers, which further increase the layout size, parasitic capacitance and power dissipation, further requiring larger buffers.

2.3 Importance of Technology Support for On-Chip Memory

The data and conclusions for this section are derived from work done by Ajay Chandna [Chandna94, Brown92b].

On-chip memory that is fast and efficient in area and power is essential to achieving performance for modern processor designs. The latency in going off-chip for the first level data cache in the Aurora III architecture is very costly to overall performance, as will be discussed later. Appropriate partitioning, the use of decoupling queues, and a fast system substrate all contribute to offsetting the lower integration levels of GaAs. Ultimately, however, it is not possible to avoid the need for dense, fast memory that is closely coupled to the computation units of a design. This section will discuss GaAs technology in terms of the characteristics that are a necessary to adequately support memory on-chip.

Subthreshold currents in MESFETs are several orders of magnitude larger than those in MOSFETs. In static RAM structures, area and power are strongly related to leakage current. Though much less attention has been focused on minimizing leakage currents than on increasing transconductance, leakage currents are as important to performance. If too many memory cells are connected to a bit line, the leakage current through the pass transistors connected to unselected memory cells (about 100nA/bit) could corrupt the data of a selected memory cell (about 20uA). The total leakage on a bit-line should be an order of magnitude smaller than the active current. Consequently, for GaAs, the number of bits that can be safely connected to a column is limited to 32. This constraint requires that a significant portion of the total RAM area be devoted to sense amplifiers and write circuitry. Table 2.4 shows how SRAM area would decrease if leakage currents could be reduced to allow more memory cells per column, thereby amortizing the column support circuitry over more bits [Oettel92]. As can be seen, for this design at 32 bits/column only 70.6% of the total chip area is consumed by the memory cells. A reduction in leakage current by only one order of magnitude would increase the percentage of area occupied by the memory cells to 92% of the total area.

In any technology, the pullup of a static RAM cell should provide just enough current to offset the leakage current of the pulldown devices. Leakage currents, therefore, also set the lower limit for cell power. In conventional GaAs DCFL processes, long, minimum-width depletion transistors are used to keep this current small. The characteristics of these devices present an area/power trade-off; for example, in the SRAM used for the Aurora III design, the highest impedance standard-threshold depletion transistor that fits in a 400um^2 cell provides much more current than is needed to offset the leakage currents. As the area of the cell is decreased, the pullup length must be decreased, increasing the power. Figure

Table 2.4 Effect of Reducing Leakage Currents on Area of 1Kx8 SRAM

Number of Bits / Column	32	64	128	256	512
Normalized SRAM Area	1.00	0.87	0.80	0.77	0.75
Cell Area Percentage of Total Area	70.6	81.6	88.4	92.1	93.8

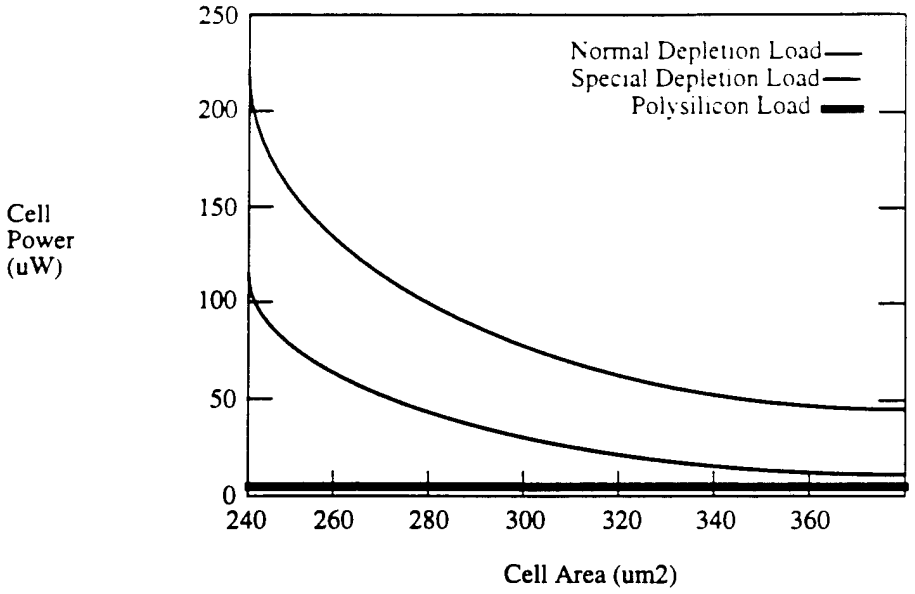


Figure 2.6 SRAM Cell Power vs. Cell Size

2.6 shows the effect of varying the pullup length (cell size) on power dissipation. This plot includes curves for a digital process pullup transistor, a special higher-threshold depletion transistor, and a polysilicon load. The polysilicon load curve was constructed assuming lightly-doped resistors, which can be located above the remaining 4 transistors, adding no additional area. As seen in the Figure 2.6, poly loads are invaluable to SRAM designs.

2.4 Summary

GaAs was chosen as the implementation technology for several reasons. First, its fast gate switching speeds and low power supply voltage seem to offer a desirable power-delay product for high-performance VLSI designs. We have designed three processors in GaAs, ranging from a very simple machine to a full-functioned superscalar design. These designs provided an opportunity to evaluate GaAs DCFL in realistic VLSI designs. Second, GaAs MESFET process technology is fairly simple, requiring many fewer fabrication steps than CMOS and offering the prospect of reasonable yields (and cost) for large designs. Other factors which can affect cost include integration density, packaging, wafer cost, and design time. Third, this technology provides a foundation for exploring a wide range of high-

performance issues which might otherwise be difficult for a university research project.

CHAPTER 3

Architectural Issues for a High Performance Floating-Point Unit

3.1 Previous Work

Previous work by the GaAs Microprocessor group at the University of Michigan has focused on both the development of a CAD environment appropriate for GaAs design and two initial implementations of the CPU architecture. The first version, called Aurora I, was based on a simple 5 stage pipeline, with 32 word register file and ALU. The chip executed 30 instructions from the MIPS Instruction Set Architecture (ISA), consisted of 60,000 transistors, and operated at 100MHz [Brown92a], [Brown93]. This chip served to drive the selection and development of many of the tools needed to design and analyze VLSI circuits in GaAs. The goal for the second generation of the CPU, Aurora II, was to study issues in high speed microprocessor architectures, including support for caches and exceptions. The chip implemented an additional 10 instructions, was comprised of 160,000 transistors (in the same area as Aurora I), and operated at 180MHz [Upton93]. In addition, timing analysis capability was added to the suite of design tools.

3.2 Aurora III System Overview

A block diagram for the Aurora III system is shown in Figure 3.1. The system is comprised of four custom GaAs chips: three logic chips and a 32K-bit SRAM used for building a 64 K-byte external data cache. The logic chips are the Integer Processing Unit (IPU), the Floating-Point Unit (FPU), and the Memory Management Unit (MMU). The IPU consists of five functional modules that operate semi-autonomously to fetch, decode, exe-

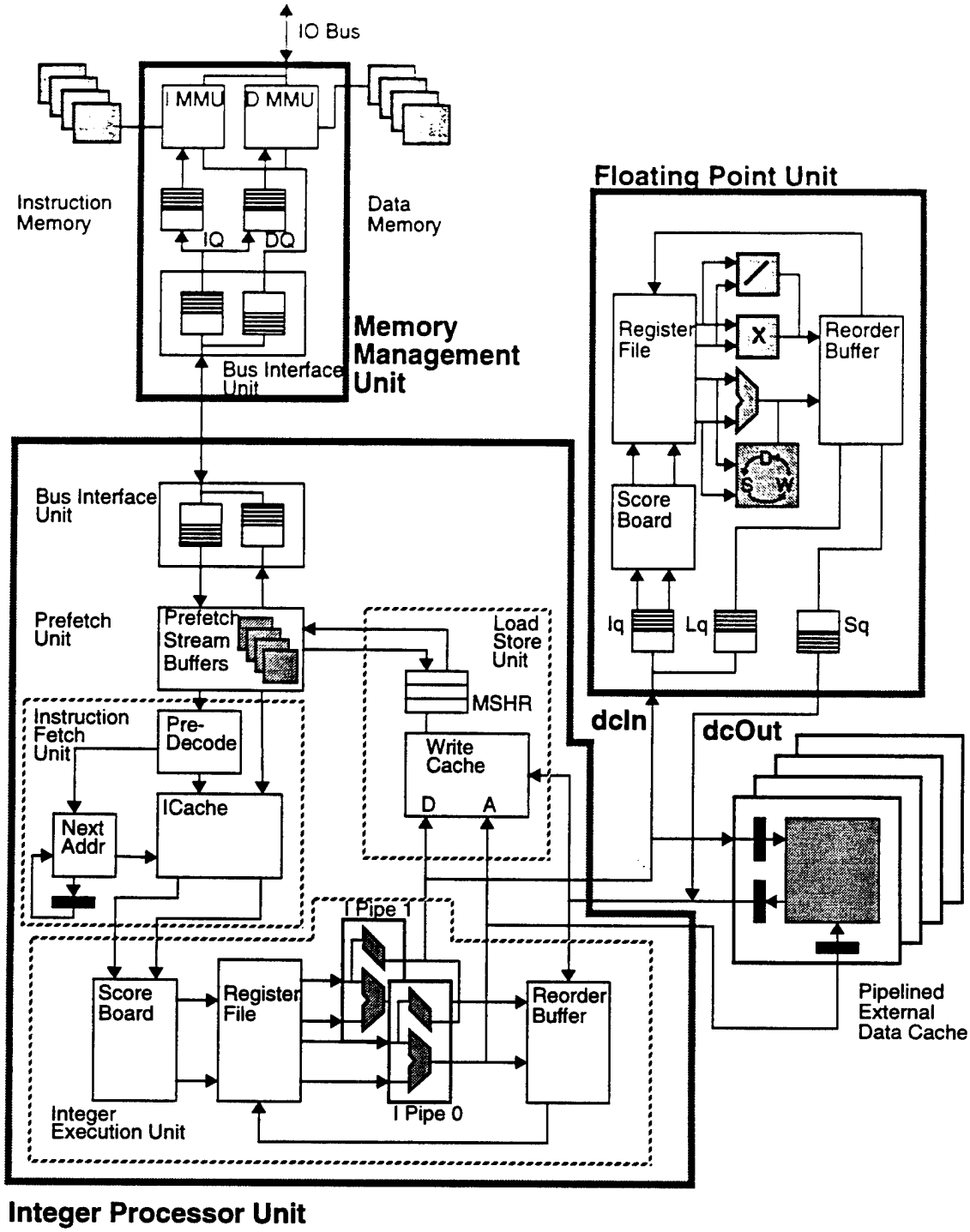


Figure 3.1 Processor Block Diagram

cute and retire instructions. The IPU is similar to the IBM-Motorola PowerPC 603 and 604 processors [Diefendorff94] in that it includes a Bus Interface Unit (BIU), an Integer Execution Unit (IEU), an Instruction Fetch Unit (IFU), and a Load Store Unit (LSU). In addition, the IPU has a dedicated Prefetch Unit (PFU) for data and instructions. The BIU

provides sustained transfer rates of 1.5 G-bytes per second over a 32-bit bidirectional bus using a collision-based protocol. The clock is sent along with the data, which allows transfers on both clock edges. The IFU fetches instructions either from a partially decoded on-chip instruction cache or from secondary memory via the BIU and MMU. An instruction miss will stall issue, but the back end of the pipeline, including active data references in the LSU, can proceed. Static branch prediction is currently supported and a dynamic scheme could be easily added to a future version of the design. The IEU contains 2 copies of the ALU and register file, which are symmetric in order to simplify issue determination. The LSU interfaces directly to a 3-cycle pipelined off-chip data cache and supports non-blocking load instructions via several miss-status holding registers. As a result, instruction issue stalls for a load miss only if a subsequent instruction needs the result of the load. Further, the LSU contains a 4-entry coalescing write cache to reduce store traffic across the BIU. Since the MMU does not reside on the same chip as the IPU, the tags of the write cache also acts as a micro-TLB, allowing store instructions to be retired quickly from the integer reorder buffer. Finally, the LSU is responsible for transferring floating-point instructions and data to the FPU. Figure 3.2 shows a more detailed view of the FPU, the characteristics of which will be discussed in greater detail below.

3.3 Simulation Methodology

The floating-point simulator developed for this study is built upon a modified version of Mike Smith's "xsim" trace-driven simulator [Smith87]. Changes were made to represent the Aurora III architecture, including dual issue of instructions, prefetching of instructions and data, an appropriate memory subsystem, and other characteristics discussed in Section 3.2. Additions to the simulator were required to implement all FPU functionality. Pixie, a program developed by MIPS Computer Systems Inc., is used to annotate an application to be studied with assembly instructions that output information about memory references and branches. The simulator executes the "pixiefied" object file and pipes the output back to the analysis routines of the simulator. Accurate information about the state of the machine is used to determine cycle and instruction counts, as well as specific

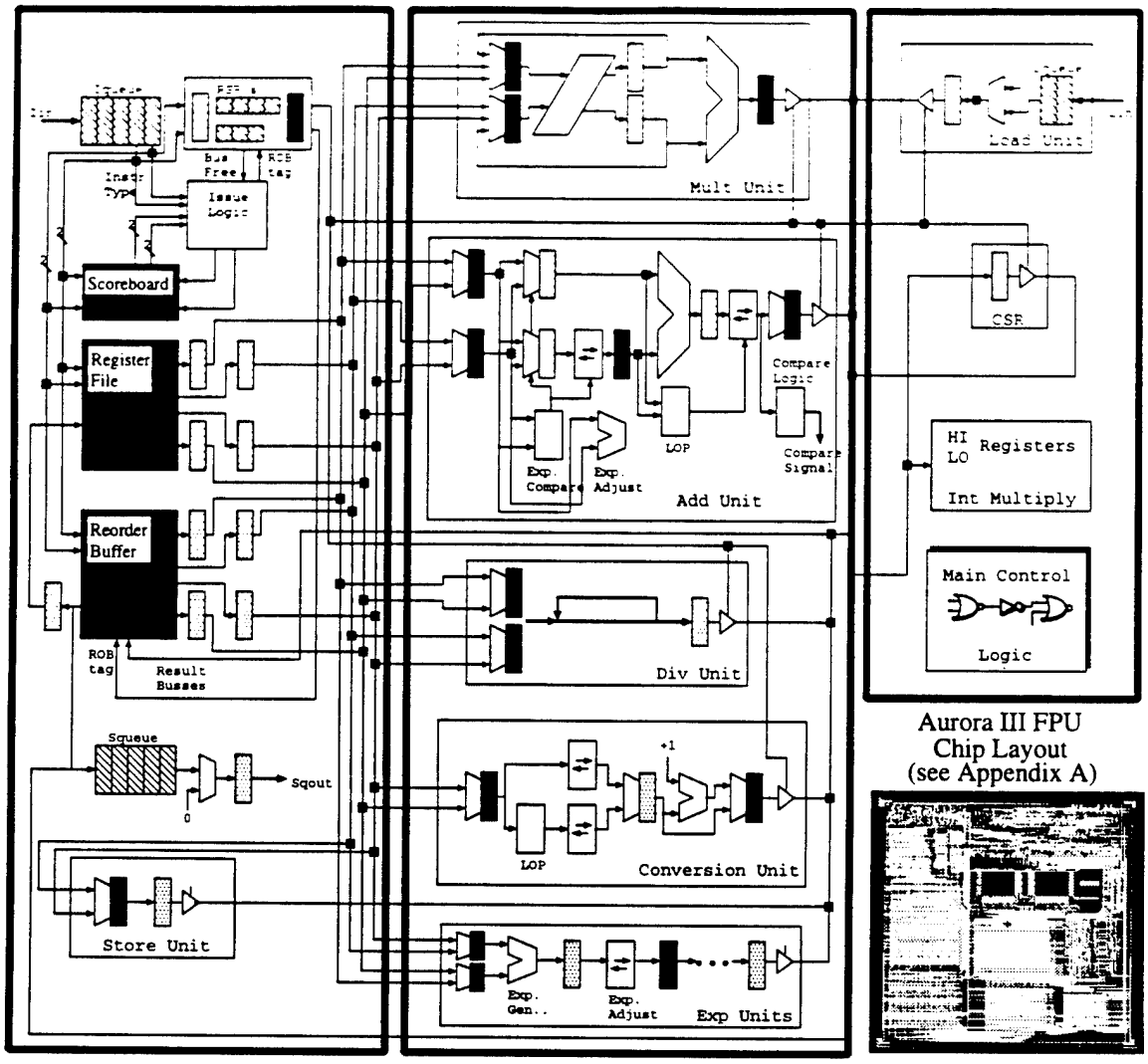


Figure 3.2 Aurora III FPU Block Diagram

information about what causes stalls (full reorder buffer, result bus conflicts, data dependencies, etc.). The SPECfp92 benchmarks are used to represent a typical scientific workload. These are comprised of 14 applications, some of which are more vectorizable than others (refer to Table 3.1). Figure 3.3 shows the dynamic instruction breakdown for each of the benchmarks. These applications are written in either FORTRAN or C, and most use either single or double precision numbers exclusively. Since many experiments were to be run and each program executed adds to the runtime, subsets of the benchmarks were often used; the rationale for choosing to use certain benchmarks will accompany the discussion of the experiment. All experiments included at least 50 million instructions and some had as many as 1 billion instructions. A larger set of sizes was used for the various IPU resourc-

Table 3.1 SPECfp92 Benchmarks

Benchmark	Comments
alvinn	Neural network used for driving an automobile - single precision - C
doduc	Monte Carlo simulation of a nuclear reactor - double precision - non-vectorizable - many subroutines - FORTRAN
ear	Use of FFT's to simulate the human ear - double precision - C
fpppp	Quantum chemistry program - double precision - difficult to vectorize - Fortran
hydro2d	Astrophysics program to solve for galactical jets - double precision - vectorizable - 48% of time spent in one subroutine - FORTRAN
mdljdp2 mdljsp2	Solves equations of motion for 500 molecules - double or single precision - vectorizable - FORTRAN
nasa7	Seven floating-point intensive tests - double precision - various matrix operations and radix-2 FFTs - FORTRAN
ora	Ray tracing through spherical and planar optics - double precision - FORTRAN
spice2g6	Analog circuit simulator - double precision - causes high data cache miss rates - FORTRAN
su2cor	Quantum physics calculation of elementary particle masses - double precision - vectorizable - 52% of time spent in one subroutine - FORTRAN
swm256	Solution of a system of shallow water equations using finite difference approximations - single precision - vectorizable - 48% of time spent in one subroutine - Fortran

Table 3.1 SPECfp92 Benchmarks, continued

Benchmark	Comments
tomcatv	Analyzes geometric domains, such as airfoils and cars - mixture of single and double precision - highly vectorizable - high data cache miss rates - Fortran
wave5	Solution of Maxwell's equations and particle equations of motion - single precision - Fortran

es and was held constant throughout all experiments, as summarized in Table 3.2.

3.4 Evaluation Criteria

Three synchronization points in the design can cause the FPU to stall the IPU. Of these stalls, the first occurs when the instruction (Iq) or load (Lq) data queue in the FPU becomes full; this in turn depends on the various constraints that can prevent issue or dispatch from the queues. (The distinction between issue and dispatch will be discussed further in the section on issue policies.) The second stall source happens whenever a branch-on-FPU instruction awaits the completion of a corresponding floating-point compare instruction. Again, a full range of internal stall conditions can delay completion of the compare instruction. The last type of stall occurs when the LSU write cache is full, and eviction

Table 3.2 IPU Resources Used for Simulation Experiments

Integer Reorder Buffer Entries	8	Number Outstanding Load References	4
Number Prefetch Buffers	16	Write Cache Entries	8
Primary Data Cache	64K	Primary Instruction Cache	4K
Secondary Data Cache	8M	Secondary Instruction Cache	8M
Data Cache Line Size	32 bytes	Instruction Cache Line Size	32 bytes
Primary Miss Latency	17 cycles	Secondary Miss Latency	60 cycles

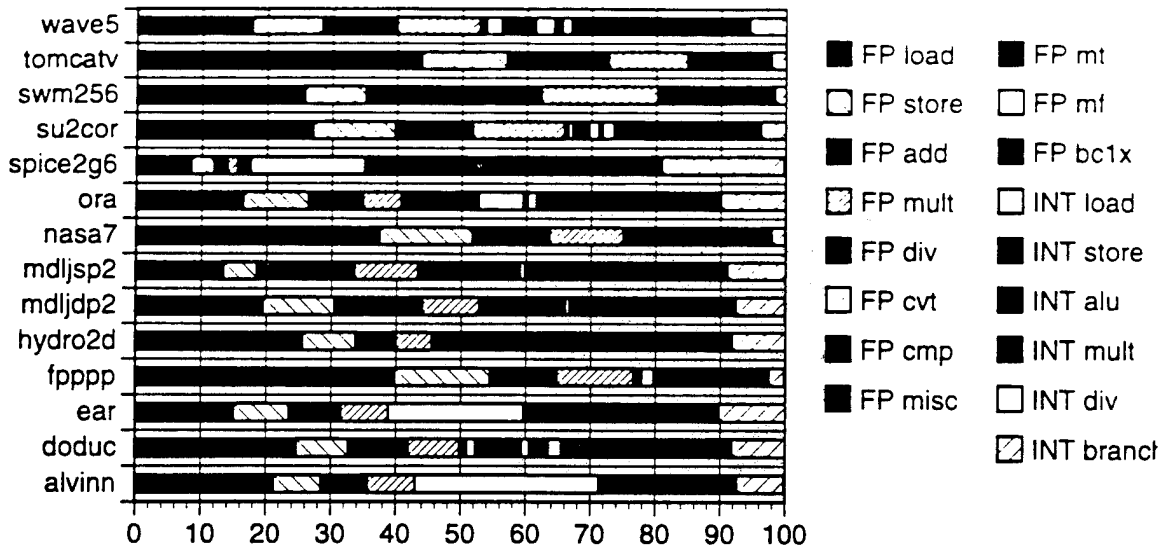


Figure 3.3 Dynamic Instruction Breakdown for SPECfp92

is prevented because one or more entries are waiting for floating-point store data. Other entries may also be locked if an outstanding cache line has yet to be returned from the secondary memory system. In both of these cases, loads and stores will stall issue to the LSU whenever it is not possible to evict a write cache entry.

The underlying mechanisms which trigger these 3 cases will be discussed in detail in the analysis that follows. To facilitate comparisons, some combination of the following metrics will be utilized:

1. *The average rate of transferring floating-point instructions to the FPU.* The interface between the IPU and FPU supports a maximum rate of 2 instructions per cycle.

2. *The frequency of issuing or dispatching more than one floating-point instruction per cycle.* An instruction is considered to have issued when all source operands are available and the instruction has been sent to a functional unit for execution. On the other hand, dispatch occurs only for an out-of-order issue policy and refers to an instruction that is transferred from the queue to the reservation station of a given functional unit; this occurs only when a source operand that is needed by an instruction is not yet available. When the data becomes available, the instruction will proceed to issue from the reservation station. For in-order issue, the maximum issue rate will be limited to 2, for reasons to be discussed below. For an out-of-order issue policy, it is

possible to have a peak issue equal to the number of functional units; maximum dispatch, and hence the maximum average throughput, will also be limited to 2.

3. *The 3 external stalls mentioned above: Iq/Lq full, branch-on-FPU, and write cache full due to outstanding floating-point stores.* These high-level stalls will be decomposed into the various conditions that can prevent issue or dispatch. Floating-point stalls per instruction (FSPI) is metric which combines all three stalls and will complement basic CPI.

4. *Average latencies and analysis of the components that comprise these latencies.* For each floating-point instruction type this will include the average latency for issue and/or dispatch (depending on the issue policy), for results becoming available, and for results writing back to the floating-point register file. The dispatch and issue latencies are measured from the time a floating-point instruction issues in the IPU, and indicate how long it takes for the instruction to proceed through the ALU and LSU and finally dispatch or issue from the floating-point instruction queue. Stall sources within the IPU include: 1) waiting for a full integer reorder buffer (floating-point instructions must also reserve an integer reorder buffer entry in order to support precise memory exceptions), 2) waiting for access to the data cache busses, and 3) waiting for a full instruction or load data queue within the FPU. The latency for results becoming available is measured to the time at which the result data is actually written into the floating-point reorder buffer. The final latency, for writing results back to the register file, is measured to the time at which an entry reaches the head of the reorder buffer; this latency indicates the length of time that data resides in the reorder buffer and infers the number of entries ahead in the reorder buffer and how long they stall write-back to the register file. For floating-point load instructions, the average latency is an indication of how often these memory references hit in the data cache. Some of the experiments will also discuss the average issue point, as opposed to the average latency to issue. The former indicates the average time taken by an instruction to reach the point in the instruction queue where issue is possible. The difference between issue point and issue latency is an indication of how long issue

is delayed due to constraints such as data dependencies and a full reorder buffer. The average latency metric will be valuable in tracking where different types of instructions spend their time, from their issue in the IPU to completion in the FPU.

5. *Utilization of the various busses shared by the IPU and FPU.* Different events contend for these busses, raising questions about whether a given bus organization can become a bottleneck for performance.

6. *Dynamic instruction counts of individual integer and floating-point instruction types.* Among other uses, this information on the frequency of instruction types can help identify ways to more efficiently allocate resources.

7. *Optimal sizes of instruction, load- and store-queues, and reorder buffer.* This can be determined dynamically for each experiment and benchmark by using a large number of entries and by keeping track of the number of entries that are actually utilized. Since these sizes should be appropriate for peak floating-point activity, information about the number of entries being utilized by a particular resource will be updated only when this resource is accessed.

8. *Resource cost.* The register bit equivalent (RBE) model of Mulder [Mulder91] is used to evaluate the resource cost of different microarchitectural features. This model uses a normalized measure of area cost which is based on the size of a 1 bit static latch. For GaAs DCFL, one static latch requires 16 transistors and corresponds to an area of 3600 square microns. Since static RAM elements are denser, a single cell is modeled as one half of a RBE. Additionally, the overhead associated with sense-amplifiers and decoding logic is represented as a percentage of the array size. The cost of various floating-point resources is shown in Table 3.3; these figures are derived from actual layout obtained during chip design.

What should be considered a meaningful improvement in performance? In light of the fact that processor performance increases by about 0.8% per week and 50% per year, any additional feature must at least keep pace. Including both design and verification time, if a feature requires a month to implement, it ought to add greater than 4% to overall per-

Table 3.3 Resource Cost in RBE Units

FPU Element	Cost in RBE
1 Floating-point Register File (32x64)	4,700
1 Scoreboard	3,600
1 Instruction Queue Entry	305
1 Load/Store Queue Entry	220
1 Reorder Buffer Entry	900
1 Reservation Station Entry (1/2 operands)	650/1030
Control Logic (25% overhead)	8,000
1 Add Unit (1 to 5 cycles)	5,000 to 1,250
1 Multiply Unit (1 to 5 cycles)	8,750 to 4,375
1 Divide Unit (10 to 30 cycles)	2,500 to 625
1 Conversion Unit (1 to 5 cycles)	2,500 to 1,250

formance. Further, if one considers the inherent inaccuracy of the various methods of predicting performance, hopefully only on the order of a few percent, a reasonable criteria might be 10%; below this amount, an idea may not be worth implementing. Clearly other issues, such as the impact on area (yield = cost), speed, and power are also equally important.

3.5 Issue Policies

In an increasing order of performance gains, issue and completion policies are as follows:

1) in-order issue, in-order completion (IOIO),

2) in-order issue, out-of-order completion (IOOO),

3) out-of-order issue, out-of-order completion (OOOO).

The first is the simplest, but achieves the worst architectural performance. In this scheme, dependency checking needs to be done only between a decoded instruction and the few instructions that are already in execution. Since results are completed in order, there is

no need for reordering prior to writing back to the register file. This simple policy stalls instruction issue whenever an instruction needs a different functional unit than that used by currently active instructions or when there is a conflict for a functional unit. The former case is necessary since the functional units may have different latencies and this policy requires instructions to complete in order. The latter case occurs in functional units that require more than one cycle to complete and allows only one active instruction at a time.

The second policy (IOOO) will stall the decode unit only when there is a functional unit conflict or a source operand has not yet been determined. A scoreboarding approach can be used to detect dependencies and conflicts. This approach can run into output dependency problems (an earlier result overwrites a later one), making it necessary to add a mechanism for reordering results, such as a reorder buffer. Also, since multiple instructions can complete at the same time, arbitration for result busses is needed. The reorder buffer is also needed to prevent erroneously repeating the execution of an instruction upon returning from an exception.

Out-of-order issue attempts to increase look-ahead capability by moving a data-dependent instruction past the decode unit and into an instruction window which resides between the decode and execute units. This issue policy increases the opportunity of finding instructions without dependencies. Out-of-order issue is subject to an additional type of data dependency, which occurs when a subsequent instruction changes a source operand of a yet-to-be issued instruction (anti-dependency). To handle this, operands are forwarded to the instruction window at the same time the instruction is sent to the window. However, since operands may not all be ready, a mechanism is needed for forwarding results from the execution units back to the instruction window. Out-of-order issue may also allow more slip between the IPU and FPU by allowing instructions to be removed from the instruction queue for three cases in addition to that of basic data-dependencies: 1) instructions which need a busy non-pipelined functional unit, 2) both instructions in the issue pair need to use the same functional unit, and 3) all result busses are busy. The apparent advantages of OOOO may in fact not be realized if some of these events occur infrequently.

Reservation stations are an alternative to a monolithic instruction window and mean

that the central window is split, not necessarily evenly, among the functional units. The registers at a reservation station can be organized in several ways. If more than one instruction at a reservation station is ready to issue, a random approach would allow any one to be selected. A simpler first-in-first-out (FIFO) ordering might be used, but with only a small performance penalty since programs have a significant amount of inherent sequential ordering. The main benefit of reservation stations is derived from the fact that instruction level parallelism (considering both integer and floating-point instructions) could support a peak issue rate of about 6 and an average of about 3 to 4 [Johnson91]. To ensure that short term demand does not stall the decode and issue unit when using a centralized window, as many as 12 source operand busses (2 per instruction) and 12 instruction window read ports might be needed. Reservation stations, on the other hand, split the instruction window among execution unit, reducing the number of global busses and instruction window required to a number closer to the average issue rate. Following are the trade-offs between using a central instruction window and using reservation stations:

1. Interconnect area and register file ports versus storage space. Reservation stations use more storage space, since they typically have more total entries than a unified instruction window. This is offset somewhat by the fact that reservation station entries need accommodate only the number of operands required by a given functional unit. Also, reservation stations can result in less global interconnect because dedicated busses are needed between the instruction window and each functional unit. Instead, fewer common tristate busses can be used to direct operands from the register file and reorder buffer to the functional units.

2. More complex issue logic versus duplicated issue logic. The central window is more complex since:

   a. It selects among a larger number of instructions.

   b. It must consider all functional unit conflicts and arbitrate among them.

   c. It must be able to issue more than one instruction per cycle.

   d. All instruction entries must be of maximum width (the instruction and 2 or 3

operands). Reservation stations allow the register width to be tailored to the needs of each functional unit.

The additional state needed for the reservation station entries can be quite large. Two entries per functional unit would contribute about 10,000 RBE's, or an additional 25%, to overall chip area. To view this another way, the resources required to implement OOOO are roughly equivalent to the difference between 2-cycle pipelined and a 5-cycle iterative multiply units. This trade-off for the multiply unit represents a 10% difference in total machine performance (see Section 3.8.1), setting a standard for cost/benefit for other architectural features, such as issue policy. In addition to the impact on area from out-of-order issue, the requirement to forward results can be expensive. This policy adds complexity in the following ways:

1. Each functional unit needs to be able to schedule result busses.
2. A tristate bypass bus is required for each load queue entry. For a 2-entry load queue, 2 additional 64 bit busses would need to be routed to each functional unit. Few things are more expensive in terms of area than large busses, since overcell routing across datapath sections tends to be quite congested. When the available routing region over a datapath cell is filled, the routing spills over into the channels, which increases the overall pitch of that datapath section and consequently the size of the entire chip.
3. A large number of comparators are needed within each reservation station entry to check the result tag on each result bus. Simulations discussed below conclude that only 2 result busses are needed. Since these are already being routed throughout the chip, there should be little increase in interconnect due to forwarding of results. If the number of reservation station entries per functional unit does not exceed 2, four comparators would be needed per functional unit, and 24 comparators overall.

A top critical path for out-of-order issue may now consist of result bus arbitration, determination of source operand availability, source bus arbitration, and functional unit arbitration.

3.5.1 IOIO versus IOOO

A baseline architectural model which uses in-order issue and in-order completion policies was chosen, and various design alternatives have been compared against this baseline. A single register instruction buffer (IRB) is used to store instructions sent by the IPU, allowing a small amount of IPU slip. Since results must complete in order, issue of two instructions to separate execution units is not allowed. This approach ensures that a later-issued instruction does not complete before an earlier issued one. Successive instructions can be issued to the same functional unit, to better take advantage of the pipelined nature of most of the units; this form of IOIO is called "pipelined" in Table 3.4. The simulator supports variable latencies for each unit, and allows any unit to be blocking (not pipelined). For example, the iterative algorithms used for division make the divide unit blocking in nature, since only one divide instruction can be active at a given time. A register scoreboard is used to determine data dependencies. The baseline clock cycle latencies for each functional unit are: load = 1, store = 1, add = 3, multiply = 5, divide = 19, conversion = 2.

Results for the baseline FPU are shown in Table 3.4. Many criteria could be used for measuring the effectiveness of an architectural feature; parts of this study will use floating-point stalls per instruction (FSPI) and overall cycles per instruction (CPI). The former is a more direct indication of the improvement of the FPU alone, whereas the latter tends to show the impact on overall performance. Even for programs where floating-point activity dominates, the majority of work is still done in the integer unit, so fairly large reductions in FSPI are needed for significant improvements in CPI. Figure 3.4 shows the ratio of integer to floating-point instructions for the benchmarks.

3.5.2 Dual Transfer and Issue of Instructions

The design of the IPU for transferring floating-point instructions and data is somewhat constrained by pin limitations. Pins that are available for transfers (without adding busses to the IPU) are the data cache input and output busses (each 64 bits), and the data cache tag bus (20 bits). These busses allow a potential transfer of two floating-point instruc-

Table 3.4 IOIO Baseline Performance

	IOIO (not pipelined)		IOIO (pipelined)	
Benchmark (50M Instructions)	FSPI	CPI	FSPI	CPI
alvinn	0.000	1.5697	0.000	1.5695
doduc	0.605	2.2460	0.374	1.8917
ear	0.593	1.6657	0.291	1.3620
fpppp	0.526	2.5826	0.290	1.6400
hydro2d	0.624	1.8912	0.480	1.6154
mdljdp2	0.840	1.8771	0.618	1.6464
mdljsp2	0.852	1.8650	0.686	1.7172
nasa7	0.836	2.0442	0.621	1.7704
ora	0.705	2.1715	0.232	2.0405
spice2g6	0.113	1.8247	0.246	1.8084
su2cor	0.646	2.1038	0.321	1.7100
swm256	0.927	2.1441	0.744	1.8062
tomcatv	0.901	2.5778	0.645	1.8724
wave5	0.828	1.9666	0.405	1.8860
Avg Arithmetic	0.547	2.038	0.422	1.738
Avg Harmonic	0.643	1.999	0.425	1.722
% Change			-33.9	-13.9

tions per cycle. The IPU cannot dual-issue a floating-point instruction and an integer instruction which both reference the data cache.

Dual issue of floating-point instructions in the FPU is an indication of the amount of parallelism available within the instruction stream. Among the constraints which can prevent dual issue are:

1. functional units conflicts,
2. true data dependencies,
3. blocking functional units that are busy,
4. result bus conflicts,

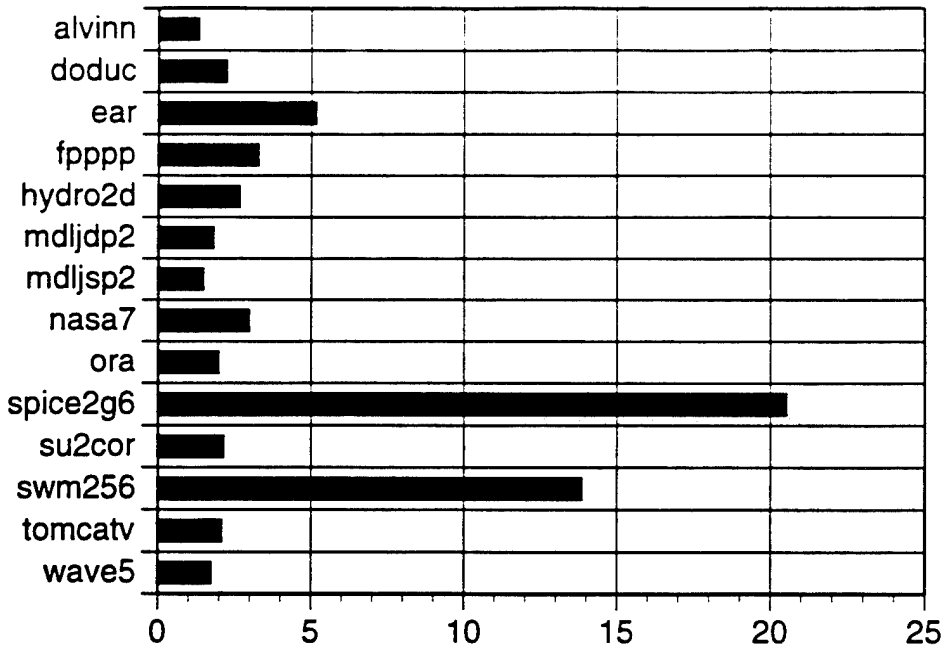


Figure 3.4 Ratio of Integer to Floating-Point Instructions for SPECfp92

5. the instruction queue containing fewer than N instructions, where N is the degree of multiple issue,

6. reorder buffer being full.

As the second and fifth columns of Table 3.5 show, there is an improvement of 15% in CPI for dual transfers and dual issue of floating-point instructions over that of single transfers and single issue. Not surprisingly, the middle 2 columns of this table show little improvement over a single transfer and issue policy, since either the peak issue rate or the peak transfer rate is limited to only one instruction per cycle. Table 3.6 shows a fairly broad range for how often dual transfers are utilized. Exactly when a dual transfer occurs and what pair of instructions is involved is as important as the frequency; this will be developed later in the discussion of instruction sequences which contain floating-point compares.

Because of the additional complexity of issuing more than 2 instructions per cycle, higher order issue was not examined. Based on the instruction traces for the benchmarks listed, floating-point loads are the most common (24.0%), while addition/subtraction in-

Table 3.5 Dual Transfers and Multiple Issue of 2 Instructions

Benchmark (50M Instructions)	Single Transfer Single Issue (CPI)	Dual Transfer Single Issue (CPI)	Single Transfer Dual Issue (CPI)	Dual Transfer Dual Issue (CPI)
alvinn	1.569	1.5686	1.5697	1.327
doduc	1.655	1.6551	1.6421	1.431
ear	1.228	1.2287	1.1706	1.005
hydro2d	1.541	1.5406	1.5321	1.287
mdljdp2	1.540	1.5396	1.5339	1.390
nasa7	1.105	1.1058	1.1059	0.953
ora	1.862	1.8624	1.8558	1.525
spice2g6	1.841	1.8409	1.8296	1.624
su2cor	1.509	1.5084	1.4949	1.231
Avg Harmonic	1.499	1.500	1.484	1.272
Avg Arithmetic	1.539	1.539	1.526	1.308
% Change from single transfer, single issue		0.0	-1.0	-15.1

Table 3.6 Dual Transfer Utilization

Benchmark	% Transfers Involving 2 Floating-Point Instructions
doduc	19.3
ear	10.8
fpppp	21.7
hydro2d	14.9
nasa7	30.6
spice2g6	5.0
su2cor	21.0
swm256	31.2
tomcatv	35.5
Avg Harmonic	15.0
Avg Arithmetic	21.1

structions (11.3%) are equally as likely as multiply instructions (9.6%). On the other hand, divide and conversion instructions occur quite infrequently (0.7% and 3.9%, respectively). The average issue rate for an IOOO policy is 1.26, as shown in Figure 3.5. This raised the question of whether there is enough parallelism to support two add units. Simulation suggests that the addition of a second add unit results in a negligible decrease in CPI (< 1%). However, since the benchmarks are compiled for a machine with only one add unit (MIPS R2000/R3000) it would be surprising to find many sequences which contain two successive instructions that both use the same functional unit; consequently, the benefit of implementing two add units should probably be examined further, in conjunction with code reordering.

Multiple issue of two instructions incurs some hardware cost, including two additional read ports on the register file and reorder buffer. However, a sense-amplifier based register file is used in the FPU; this design should allow additional read ports without too large of a penalty in speed or area. The reorder buffer will have a small number of entries so its performance will not suffer much from having two additional read ports. The instruction queue also needs an additional read port to allow access to the instruction immediately below the head of the queue. Additional source operand busses are needed, as well as somewhat more complex control for instruction decoding and issue. Extra write tag ports are re-

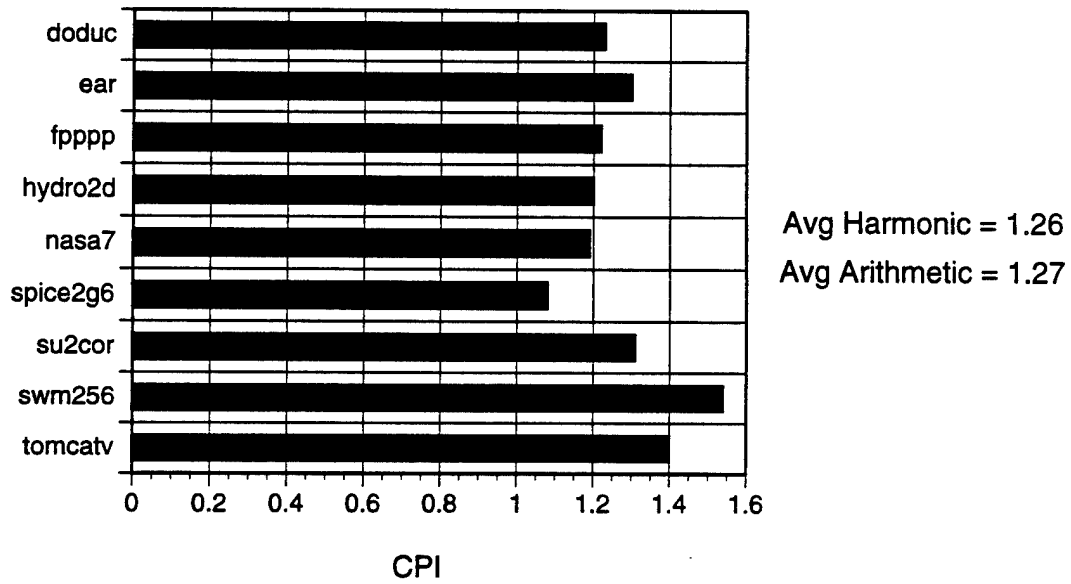


Figure 3.5 Issue Degree for IOOO Policy

quired for the reorder buffer and also for the tag lookup table. Many of these issues will be revisited in Section 3.11.1, which discusses the merits of a simpler design for the FPU.

3.5.3 IOOO versus OOOO

Out-of-order issue changes the issue paradigm. In out-of-order issue, instructions are allowed to dispatch from the instruction queue to a reservation station for the appropriate functional unit. The constraints for dispatch are now:

1. two, one, or no valid Iq entries,
2. free reservation stations for the required functional unit, and
3. free reorder buffer entries, which must be reserved at dispatch in order to retain the in-order sequence of the program.

To limit the number of reorder buffer and register file ports, the degree of dispatch is limited to two. Note that dispatch can occur in spite of conditions which would stall an IOOO policy, including the operands being unavailable, both instructions needing to use the same functional unit, or the instruction needing a blocked functional unit. The actual issue of an instruction now occurs within a reservation station and the constraints are:

1. a reservation station entry is valid, meaning that all necessary operands have been forwarded to the entry,
2. a non-pipelined functional unit is not blocked with a prior instruction, and
3. a result bus is available for when the instruction completes.

An upper bound on the performance gained by using an OOOO policy for both integer and floating-point instructions can be derived by considering both the percentage of cycles that a dual issue occurs and the percentage of instructions that are dual issues. Figure 3.6 shows these figures for an IOOO Aurora III model, as well as for a DEC 7000 AXP system (which is not an out-of-order issue machine) [Cvetanovic94]. Assuming middle range figures, such as for "fpppp" for the Alpha, dual issues occur 10% of the time and account for 40% of all instructions. For 100 issues, it means there are 40 dual issues, 60 single

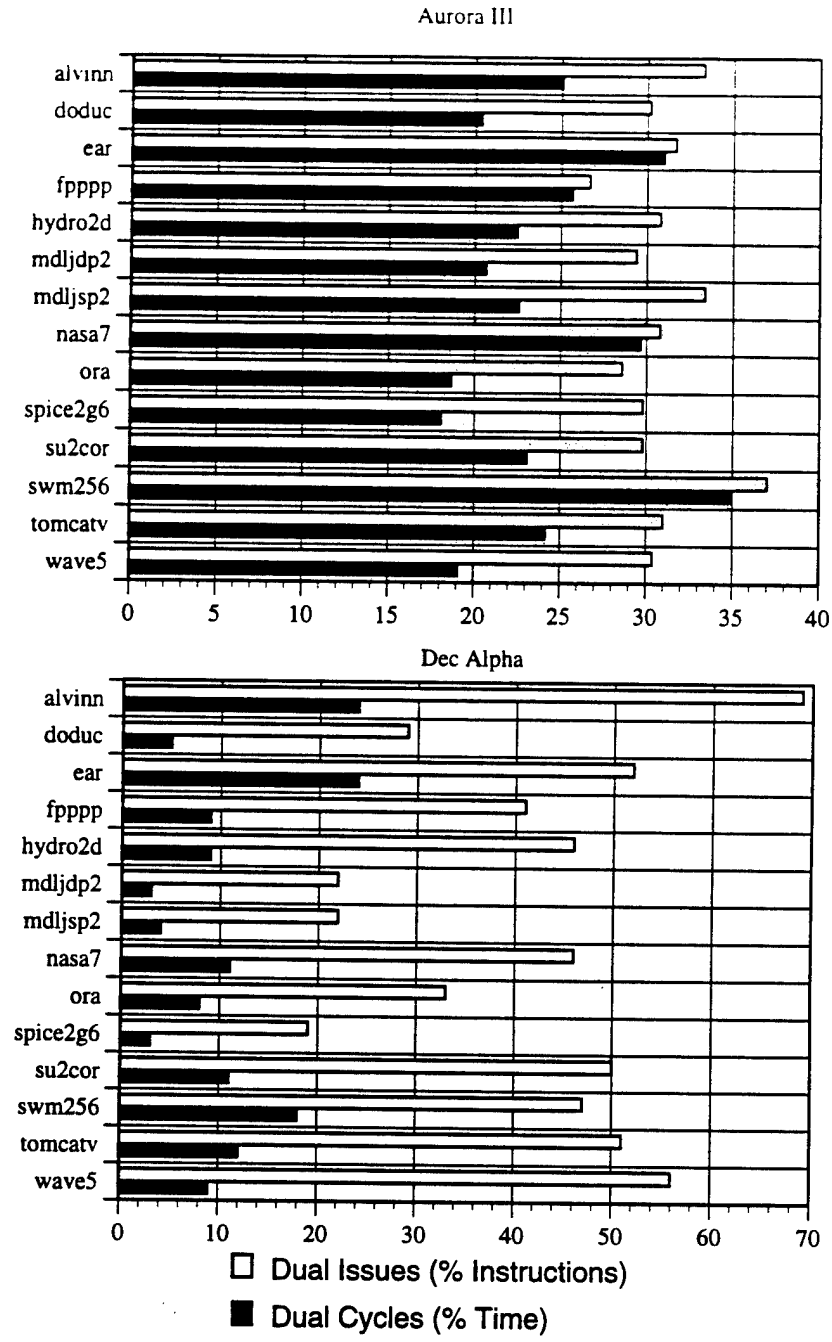


Figure 3.6 Percentage of Dual Issue Instructions and Cycles

issues, and 400 cycles overall. Further, this means that there are 300 NOP cycles due to stalling. For the best case scenario, if an OOOO policy turns all of these single issue cycles into dual cycles, there would be a savings of 30 cycles, or 7.5%. This analysis assumes that NOP stall cycles are relatively constant, being caused by memory system latencies. Table 3.7 shows this upper bound across all of the benchmarks. The Alpha improvements

are quite low since very few cycles actually issue 2 instructions, suggesting long memory latencies. The Aurora III estimates include dual issue of both integer and floating-point instructions; however, an OOOO policy will be considered only for the issue of floating-point instructions. Consequently, there will be fewer dual issue cycles and the performance benefit will be lower than suggested by Table 3.7.

For this initial set of experiments, all resources were set to a maximal amount (queues and reorder buffer having 20 entries and reservation stations having 15 entries), in order to identify an upper bound on the performance difference between the two issue polices. Some interesting results, evident from the initial experiments, are summarized in Table 3.8 to Table 3.10. First, the OOOO policy actually has worse performance than the

Table 3.7 Upper Bound for OOOO Performance Improvement

Benchmark (50M Instructions)	Aurora III (% gain)	DEC 7000 Alpha (% gain)
alvinn	25.138	5.391
doduc	23.575	6.121
ear	33.396	11.077
fpppp	35.277	6.476
hydro2d	25.276	5.283
mdljdp2	24.854	5.318
mdljsp2	22.532	7.091
nasa7	33.364	6.457
ora	23.342	8.121
spice2g6	21.319	6.395
su2cor	27.208	5.500
swm256	29.797	10.149
tomcatv	26.932	5.765
wave5	21.864	3.536
Avg Harmonic	26.043	6.132
Avg Arithmetic	26.705	6.620

IOOO policy, for 3 of the 4 benchmarks. Examining the 3 high-level stall sources for the FPU, it is clear that the most significant source involves branches that must wait for floating-point compares. Each of the benchmarks, except for spice2g6, see approximately one additional cycle of latency before a compare instruction issues from a reservation station (OOOO) versus issuing directly from the queue (IOOO). In modeling the OOOO policy, an initial assumption was made that every instruction must first pass through a reservation station prior to issue. In other words, even if all necessary operands are ready, the instruction is first dispatched to the reservation station of the appropriate unit. This approach introduces the unnecessary additional cycle of latency evident in Table 3.10. An obvious solution would be to allow issue to proceed directly from the instruction queue if both operands are valid. This adds a bit of complexity to the dispatch logic and also an additional mux input for each operand to the input stage of each functional unit. Not all types of instructions suffer from this additional cycle of latency, as seen in Table 3.11 for the "hydro2d" benchmark. Still, it is probably more efficient to allow this fast bypass of reservation stations for all instructions than to recognize only a few cases. The impact of doing so is shown in Figure 3.7. The average drop in CPI across the benchmarks is a modest 1.2%, though some applications see as much as 5.4% improvement.

While an IOOO policy sees a delay from the time the reorder buffer is written to when a result is retired, the effect is even worse for an OOOO policy, which explains in part why the measured gains in performance for OOOO are quite small (or even worse). Table 3.11 shows this same behavior for instructions in addition to floating-point com-

Table 3.8 Issue Policies (IOOO vs OOOO)

Benchmark (50M Instructions)	IOOO (CPI)	OOOO (no fast issue) (CPI)	OOOO (fast issue) (CPI)
ear	1.005	0.963	0.951
fpppp	1.024	1.030	1.034
hydro2d	1.287	1.347	1.308
spice2g6	1.624	1.644	1.632
Avg Harmonic	1.189	1.1905	1.178
Avg Arithmetic	1.235	1.2461	1.231

Table 3.9 High Level FP Stall Sources

Benchmark (50M Instructions)	Iq/Lq Full	bc1x Wait	Write Cache Eviction
ear (IOOO)	9.97	9.99	0.00
(OOOO)	3.02	13.47	0.00
fpppp (IOOO)	1.31	1.43	0.01
(OOOO)	1.42	1.64	0.01
hydro2d (IOOO)	1.90	21.92	0.00
(OOOO)	0.70	35.77	0.00
spice2g6 (IOOO)	0.00	17.76	0.00
(OOOO)	0.00	18.90	0.00

Table 3.10 Latencies for Floating-Point Compare Instructions

Benchmark (50M Instructions)	avg latency	avg ROB ready	avg issue	avg dispatch
ear (IOOO)	8.175	8.174	5.175	
(OOOO)	10.275	9.135	6.136	4.129
fpppp (IOOO)	12.712	11.666	8.689	
(OOOO)	14.421	12.495	9.515	4.061
hydro2d (IOOO)	11.032	10.396	7.396	
(OOOO)	12.656	11.057	8.057	4.353
spice2g6 (IOOO)	14.579	14.579	11.579	
(OOOO)	15.640	14.171	11.171	4.481

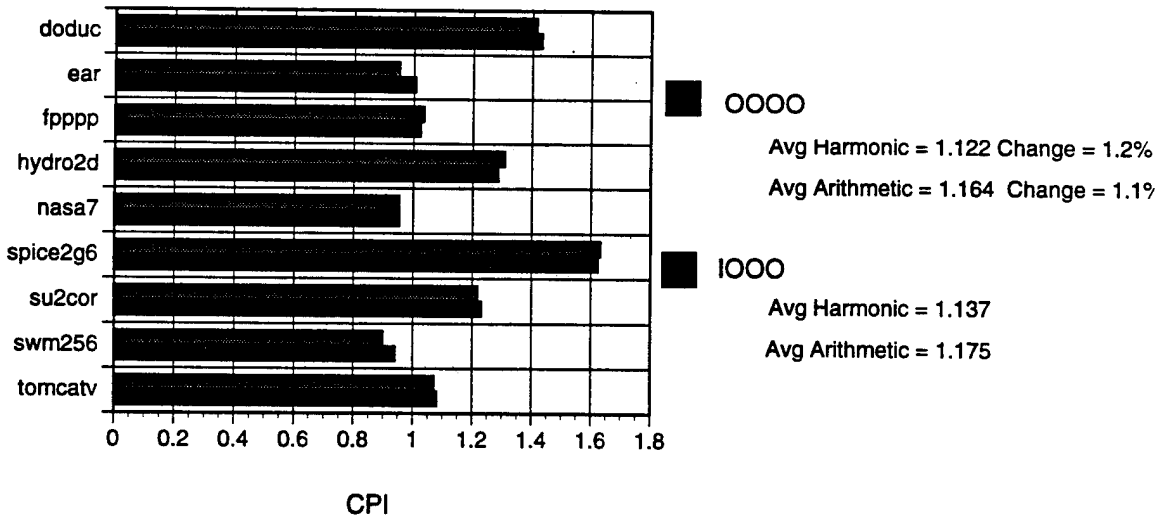


Figure 3.7 Comparison of IOOO and fast OOOO Policies

Table 3.11 Avg Latencies of Various Floating-Point Instructions for Hydro2d

Functional Unit	Instruction count	% total instructions	Avg latency	Avg ROB ready	Avg issue	Avg dispatch
LOAD (IOOO)	12920135	45.90	16.643	13.440	12.256	
(OOOO)			17.850	9.484	8.485	6.630
STORE (IOOO)	4299870	15.30	16.496	13.099	11.972	
(OOOO)			17.707	10.263	9.325	7.135
ADD (IOOO)	3124150	11.00	19.899	16.754	13.481	
(OOOO)			20.894	14.008	11.094	7.658
MULT (IOOO)	2736150	9.70	19.210	14.900	12.752	
(OOOO)			19.869	12.663	10.745	6.953
LOADXTRA (IOOO)	2268717	8.60	8.932	8.315	7.174	
(OOOO)			10.694	7.788	6.791	4.800
COMPARE (IOOO)	2268717	8.00	11.032	10.396	7.396	
(OOOO)			12.656	11.057	8.057	4.353
DIV (IOOO)	434402	1.50	41.043	41.041	21.489	
(OOOO)			38.902	37.596	18.638	8.514
CONV (IOOO)	42	0.00	6.643	6.643	4.643	
(OOOO)			7.571	6.571	5.619	4.619

pares. However, only store instructions share the same constraint as compares in needing to wait until they reach the head of the reorder buffer before their results can be used (the reason is discussed in Section 4.5); other instructions can bypass their results in the same cycle the reorder buffer is being written. Because more instructions have moved past the instruction queue, more entries in the reorder buffer are necessary, as seen in Figure 3.8 These instructions, and their corresponding latencies for completion, delay the time at which compares and stores can be retired. Having more such entries precede a compare simply makes the compare wait longer to complete. The ability to retire more than one reorder buffer entry per cycle would help alleviate this problem, but might require an addition register file write port. For GaAs sense-amplifier-based RAM's, write ports are expensive compared to read ports. Some floating-point instructions, such as compares and stores, do not produce results that are written to the register file, so it should be possible to allow two reorder buffer entries to be retired if one or more are these types of instruction. Consequently, an OOOO policy not only adds state in the form of reservation stations but also by re-

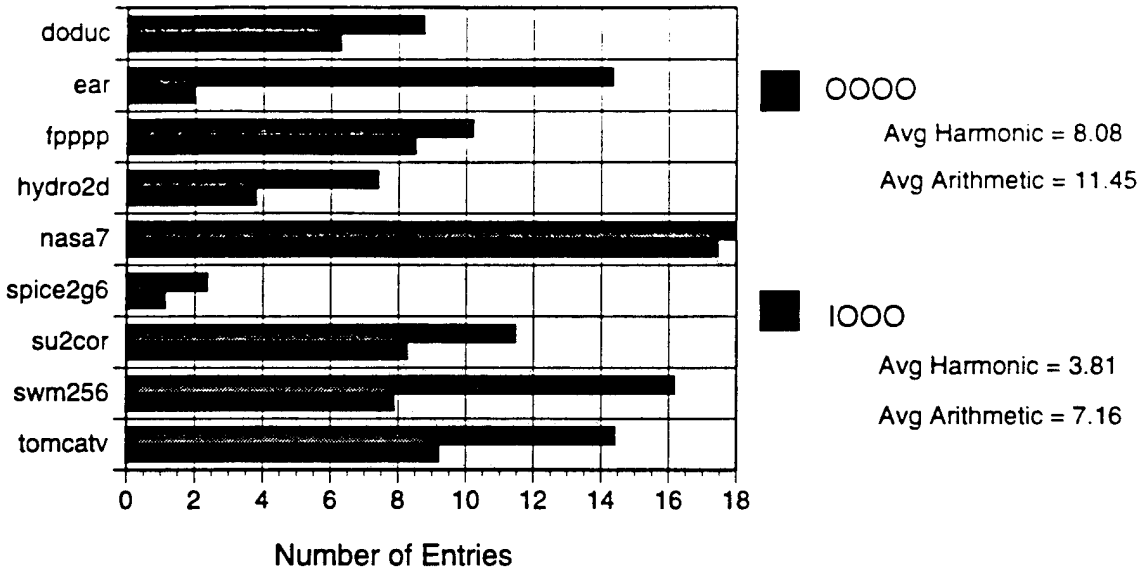


Figure 3.8 Reorder Buffer Entries Needed for IOOO and OOOO Policies

quiring a larger reorder buffer.

Another effective approach to minimizing stalls due to floating-point branches involves something external to the FPU, the memory system. Even for data cache hits, a 3 cycle latency is costly, underscoring the importance of a technology supporting large on-chip memory structures. Some of this latency might be hidden by better compiler support. The Aurora III architecture allows only one load to be issued per cycle. Dual issuing loads might be justified, though doing so may require either a dual-ported or interleaved first-level data cache. Using both hardware and software compiler approaches, a processor should issue load instructions to the memory system as soon as possible. The above analysis assumes a 17-cycle latency for a cache miss, which is probably optimistic. The overhead associated with the Aurora III collision-based bus interface unit was found to add 6 to 7 cycles to the overall latency. Collisions occurred more frequently than anticipated, which increased the overall cost of handling a memory reference. Some of these benchmarks thrash the data cache, as is evident in the average load latencies shown in Table 3.12. The latency for a load which hits in the cache should be approximately 6 cycles (recall that these latencies are measured from when the instruction issues), whereas for a cache and prefetch miss the latency would be 20 cycles. When the average load latency is in the range of 8 to 11 cycles, as seen for most entries in the tables the majority of loads are hitting in the cache.

A final comparison of IOOO and OOOO policies contrasts the average issue rate of each. For an IOOO policy, the peak issue rate is 2, whereas for an OOOO policy the peak can be higher since issue is possible from the reservation stations at each functional unit. The average issue rate for both policies cannot exceed 2, since the peak dispatch for OOOO is 2. In order for one policy to surpass the other, the average issue rate must be higher. Table 3.13 shows the issue rate as a function of total issues and number of issues per cycle. The latter parameter indicates how often issue occurs, while the former makes the nature issue evident. The number of issues per cycle is slightly higher for the OOOO policy, however the average issue rate per issue is lower. The product of the two is represented in the third column; there is little if any difference between the two policies. This result correlates well with the very small change in CPI that has been observed; an OOOO policy simply does not succeed in increasing the number of dual issue cycles.

The discussion up to this point has focused on the upper bound on performance afforded by out-of-order issue and has assumed a large resource budget. Consequently, none of the experiments showed a significant stall component due either to full queues or write cache eviction. Only about half of the benchmarks were limited by floating-point branch stalls. When the resources are reduced to a more reasonable level (Iq = 6, reorder buffer =

Table 3.12 Breakdown of Average Load Latencies (IOOO Baseline)

Benchmark (50M Instructions)	Avg Latency (cycles)	Avg Reorder Buffer Ready (cycles)	Avg Issue (cycles)	Avg Issue Point (cycles)
doduc	11.744	7.956	6.901	5.897
ear	11.242	9.136	7.992	7.377
fpppp	11.130	6.542	5.425	4.541
hydro2d	13.890	10.779	9.623	7.866
nasa7	16.397	10.889	9.761	8.505
spice2g6	14.371	13.367	12.271	9.116
su2cor	16.198	11.895	10.772	9.271
swm256	15.993	11.594	10.394	9.096
tomcatv	17.924	13.139	11.973	10.272

Table 3.13 Issue Rate for IOOO and OOOO Policies

Benchmark (50M Instructions)	Issue Rate per Issue	Issues per Cycle	Issue Rate per Cycle
doduc (IOOO)	1.23	0.34	0.42
(OOOO)	1.13	0.37	0.42
ear (IOOO)	1.30	0.37	0.49
(OOOO)	1.23	0.41	0.51
fpppp (IOOO)	1.22	0.65	0.80
(OOOO)	1.17	0.68	0.80
hydro2d (IOOO)	1.20	0.36	0.44
(OOOO)	1.12	0.38	0.42
nasa7 (IOOO)	1.19	0.78	0.93
(OOOO)	1.09	0.85	0.92
spice2g6 (IOOO)	1.08	0.06	0.07
(OOOO)	1.01	0.07	0.07
su2cor (IOOO)	1.31	0.39	0.52
(OOOO)	1.15	0.45	0.52
swm256 (IOOO)	1.54	0.55	0.85
(OOOO)	1.23	0.71	0.87
tomcatv (IOOO)	1.40	0.59	0.82
(OOOO)	1.24	0.67	0.82

8, reservation station entries = 2), the effect on CPI for both policies is still small, as summarized in Table 3.14. There is only a modest impact on performance due to limiting resources; this is not surprising since the sizes of queues, reorder buffer, etc., were derived from the simulations. The difference between the two issue policies is less than 3%. A question arises about whether the stall profile for the baseline architecture is the same as for the larger one, considering the change in CPI is small. Table 3.15 shows that the profiles are similar, with some increases in queue and floating-point branch stalls for the baseline FPU.

3.5.4 Reservation Station Selection Policy

For an OOOO policy, a choice must be about how to select an instruction to issue from a reservation station when more than one instruction is ready. A simple approach would be to examine instructions in a first-in first-out (FIFO) ordering, which is easy to im-

Table 3.14 Resource Allocation for IOOO and OOOO Policies

Benchmark	IOOO Large (CPI)	IOOO Base (CPI)	Speedup	OOOO Large (CPI)	OOOO Base (CPI)	Speedup
doduc	1.4309	1.4836	1.0368	1.4136	1.4767	1.0446
ear	1.0049	1.0204	1.0154	0.9507	0.9546	1.0041
fpppp	1.0236	1.0362	1.0123	1.0338	1.0920	1.0563
hydro2d	1.2866	1.3653	1.0612	1.3077	1.3538	1.0353
nasa7	0.9527	1.0368	1.0883	0.9516	1.0913	1.1469
spice2g6	1.6238	1.6464	1.0139	1.6322	1.6325	1.0002
su2cor	1.2309	1.2859	1.0447	1.2170	1.2986	1.0671
swm256	0.9404	1.0587	1.1259	0.8991	1.0477	1.1653
tomcatv	1.0811	1.2813	1.1852	1.0716	1.2916	1.2053
Avg Harmonic	1.137	1.212	1.062	1.122	1.215	1.076
Avg Arithmetic	1.175	1.246	1.065	1.164	1.249	1.081

plement via the head pointer of a queue. An alternative might involve choosing at random one instruction from the pool of all ready instructions. This scheme may be slightly more difficult to implement since all entries in a reservation station would need to be examined concurrently. The logic would need to identify those instructions that are ready and then select one instruction to issue. It is possible that this additional logic would contribute adversely to the path-length of overall instruction issue, though the actual number of reservation stations needed per functional unit would be quite small. Table 3.16 summarizes the results for both of these polices and shows that there is very little difference between the two. In fact, for 4 benchmarks shown a FIFO prioritization yields slightly better performance, though the difference is not significant. This result is consistent with intuition because the instruction whose result is needed the most is the oldest active instruction, which is the first instruction in the queue.

Table 3.15 Resource Allocation and High Level Stall Sources (IOOO Policy)

Benchmark (50M Instructions)	FSPC (total) (% cycles)	Iq/Lq Full (% cycles)	bc1x Wait (% cycles)	Write Cache Eviction (% cycles)
doduc (large)	22.5	2.6	19.9	0.0
(base)	26.4	8.9	17.5	0.0
ear (large)	20.0	10.0	10.0	0.0
(base)	21.2	10.2	11.0	0.0
fpppp (large)	3.7	1.3	1.4	0.9
(base)	5.4	3.3	1.5	0.5
hydro2d (large)	33.8	1.9	31.9	0.0
(base)	40.1	7.8	32.2	0.0
nasa7 (large)	12.6	9.8	2.8	0.0
(base)	23.4	20.8	2.6	0.0
spice2g6 (large)	17.8	0.0	17.8	0.0
(base)	18.8	0.0	18.8	0.0
su2cor (large)	12.4	3.2	8.3	0.9
(base)	20.8	15.3	5.3	0.1
swm256 (large)	19.1	12.0	6.5	0.6
(base)	32.0	28.5	3.3	0.2
tomcatv (large)	10.8	1.9	8.9	0.0
(base)	34.4	26.9	7.5	0.0

Table 3.16 Reservation Station Entry Selection Policy

Benchmark (50M Instructions)	FIFO (CPI)	Random (CPI)	Speedup
ear	0.9497	0.9507	1.0011
fpppp	1.0341	1.0338	0.9997
hydro2d	1.3057	1.3077	1.0015
spice2g6	1.6325	1.6322	0.9998
Avg Harmonic	1.177	1.178	1.001
Avg Arithmetic	1.230	1.231	1.001

Table 3.17 Branch-on-FPU Stalls (IOOO Policy)

Benchmark (50M Instructions)	% of Total Cycles
doduc	19.9
ear	9.9
hydro2d	31.9
mdljdp2	41.8
mdljsp2	45.6
ora	8.9
spice2g6	17.8
su2cor	8.3
swm256	6.5
tomcatv	8.9
wave5	34.1

3.6 Improving The Latency of Floating-Point Compare Instructions

Table 3.17 and Table 3.18 show the importance of synchronization stalls due to floating-point branches. Table 3.17 identifies the benchmarks which have the highest percentage of compare instructions. Table 3.18 shows the average latencies for floating-point-compares in these benchmarks, including a breakdown of where the latency occurs ("mdljsp2" is not included, since it has similar behavior to "mdljdp2"). The difference between

Table 3.18 Compare Latencies for High Branch-Stall Benchmarks

Benchmark (50M Instructions)	Avg Stall Cycles per Floating-Point Branch	Avg Latency	Avg ROB Ready	Avg Issue (Issue takes place)	Avg Issue Point (Issue can take place)
doduc	8.97	11.155	9.671	6.691	4.726
hydro2d	9.70	12.015	10.378	7.378	6.272
mdljdp2	8.61	10.173	8.642	5.642	3.428
spice2g6	14.60	15.562	14.562	11.563	11.035
wave5	10.56	14.608	13.281	10.281	7.408

the average number of stall cycles per floating-point branch (column 2) and the average latency for a compare instruction (column 3) can be explained by instructions which intervene between the compare and branch; on average, 1.4 instructions fall between the compare and branch across these five benchmarks. Latency to the issue point is an indication of how significant data cache misses are for a particular benchmark; "spice2g6" has both the highest issue point latency and the highest data cache miss rate. In the "wave5" benchmark, load and compare issue are further delayed by prior instructions in the queue. These latencies are quite large; reducing them is an important part of optimizing and FPU design.

The Aurora III design allows only one floating-point compare to be outstanding at a time to simplify the interface between the IPU and FPU. Allowing more than one compare instruction to be active at a time might seem to offer the potential of reducing branch-on-FPU stalls. However, a second compare is seldom encountered while a first one is still active. This is explained by the fact that there is a one-to-one pairing of branch-on-compare with the actual compare; issue of the branch will stall until the compare is resolved.

Table 3.10 shows that about four cycles of latency are due to the time required for the floating-point instruction to reach the instruction queue in the FPU. The IPU has pipeline stages for register fetch (RF), alu execution (ALU), and load-store unit execution (LSU). Contention for the data cache bus can add more latency but this tends to happen infrequently (see Section 3.7.4). Dispatch from the queue is also constrained by the three conditions listed in Section 3.5.3; however, since in this set of experiments there are a large number of reservation stations and reorder buffer entries, it is unlikely that dispatch will stall for these reasons.

One could move the transfer of floating-point instructions up in time to the same point at which issue occurs in the IPU, thereby saving the two cycles that correspond the ALU and LSU pipe stages. In this approach, one would have to add dedicated busses between the IPU and FPU for instructions and data (three 64 bit busses) to avoid stalling the front end of the machine. The current Aurora III design decouples the front end of the IPU pipeline (IC, RF, ALU) from the longer-latency backend (LSU). When a floating-point in-

struction transfer is stalled while a higher priority event takes place (such as a cache fill), the issue of integer instructions is not inhibited. As will be discussed later, a desire to limit pin count was a key factor in choosing the current organization. However, it is not clear that moving the transfer of floating-point instructions forward in time would necessarily result in lower latencies for compares. The merit of an early transfer point depends upon how often a floating-point compare is preceded by a floating-point load. A load instruction must still pass through the RF and ALU stages, and will see a three-cycle latency for data to be returned from the off-chip pipelined data cache (assuming a cache hit). If this pattern occurs often, the latency for compares will be roughly the same as in the current system, regardless of an early transfer. For the benchmarks which are limited by floating-point branches, examining the characteristics of the most commonly occurring compares reveals the following sequences:

```
1:
    lwc1   $f4
    lwc1   $f5          <= $f4 and $f5 comprise a single double-precision register
    integer op
    mov.d $ f0,$f4 or sub.d $f0,$f2,$f4
    cmp    $f0,$f2
    xx
    bc1x
2a:
    lwc1   $f4
    lwc1   $f5
    integer op
    cmp    $f4,$f10
    xx
    bc1x
```

Table 3.19 Common Compare Instruction Sequences

Benchmark	Compare Sequence	% Total Compares
doduc	1	44.5
hydro2d	1	18.0
	2A	24.0
	3	24.0
mdljdp2	1	37.8
	3	37.8
wave5	2A	40.0
	2B	21.0
	3	29.0
spice2g6	2A	94.5

2b:

```
lwc1   $f4
lwc1   $f5
integer op
integer op
cmp    $f4,$f10
xx
bc1x
```

3:

```
integer op
cmp    $f8,$f2
integer op
bc1x
```

Table 3.19 shows which sequences are prevalent for the five benchmarks in question. The first important observation about these sequences is that they fall into three cate-

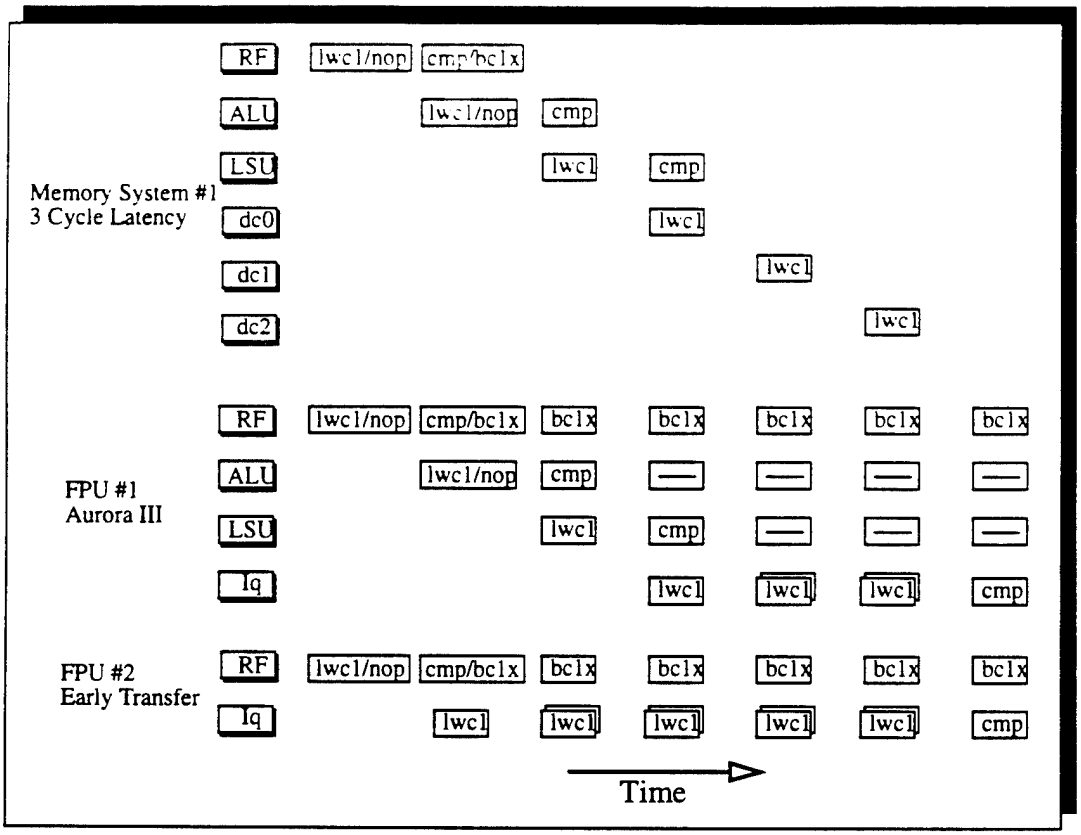


Figure 3.9 Timing For Aurora III Memory System and Early Instr. Transfer

gories. Sequences in group 1 have two intervening instructions between the load and compare. One of these instructions is a floating-point operation the source of which is the load and the result of which is used by the compare. Group 2 sequences have either one or two intervening integer instructions. The group 3 sequences are characterized by the fact that the operands needed by the compares are readily available and no issue stalls occur due to data dependencies. For all groups, on average 1.08 instructions fall between the load and compare.

Note that the sequence of a load followed immediately by a compare does not appear; this case cannot occur in the MIPS architecture because of the load delay slot. Figure 3.9 contrasts the way a load-compare sequence progresses though the current Aurora III architecture and a design which moves floating-point transfers forward in time. The compare cannot be issued in either case until the fifth cycle; the early transfer design holds the compare in the instruction queue longer. Having more instructions between the load and compare delays the arrival of the compare at the instruction queue, and increases the chance of

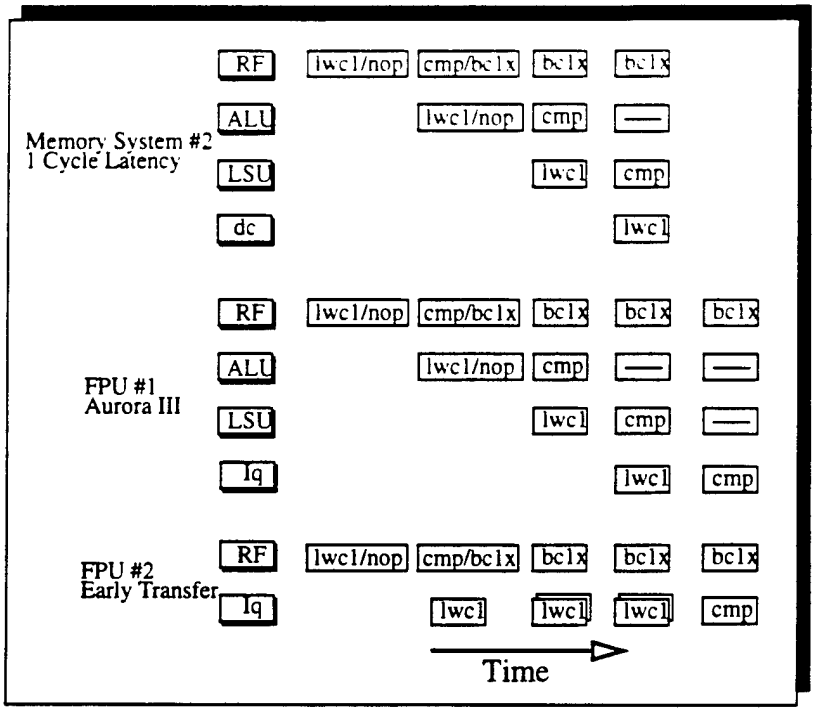


Figure 3.10 Timing For 1-Cycle Memory System and Early Instr. Transfer

benefitting from an earlier instruction transfer. With two intervening instructions in an early transfer design, the compare reaches the queue one cycle before the load issues. In the current design, the compare would reach the queue one cycle after the load issues, thus costing one cycle of latency. With three or more intervening instructions, the penalty is the two cycles corresponding to the instruction passing through the ALU and LSU pipe stages.

Compare latency can be improved in several ways in order to reduce branch-on-FPU stalls. Much of the above discussion depends on the Aurora III architecture and a three-cycle latency for a cache hit. A technology which supports a large on-chip single cycle cache would benefit more from an earlier transfer of floating-point instructions (see Figure 3.10). Compilers could also improve the performance of early floating-point instruction transfer by trying to schedule more instructions between the load and compare to better fit the parameters which affect FPU stall cycles. Table 3.20 summarizes the results for three ideas of early instruction transfer, faster primary data cache, and better code scheduling. For single issue, less distance is needed between the load and compare to benefit from an early transfer point. As the table shows, a faster memory system is also a requirement for

Table 3.20 Branch-on-FPU Stall Cycles for Different Organizations

	Single Issue			Dual Issue				
FPU Configuration	Interval of 1 instr.	Interval of 2 instr.s	Interval of 3 instr.s	Interval of 1 instr.	Interval of 2 instr.s	Interval of 3 instr.s	Interval of 4 instr.s	Memory System
#1 (Aurora III)	3	2	2	5	4	4	3	3 Cycle Latency
#2 (Fast Transfer)	3	1	1	5	4	4	3	
#1 (Aurora III)	2	2	2	3	2	3	2	1 Cycle Latency
#2 (Fast Transfer)	1	0	0	3	2	2	1	

achieving the greatest benefit of an early transfer. A dual issue processor needs more useful instructions to be scheduled between the load and compare to have the desired effect, because it retires them in pairs. An early transfer point results in the best performance, but only if supported by both better code scheduling and a faster memory system. These options also impact the number of entries needed for the instruction queue, as shown in Table 3.21. In these cases, the early transfer policy fills the queue faster than instructions can be issued from it. The lower right design points are the best in each of the 4 configurations for memory system and issue degree.

An optimistic upper bound on the performance gained by these changes would assume that compare latency is dominated by the speed of the add unit and not by the delay

Table 3.21 Average Instruction Queue Entries for Different Organizations

	Single Issue			Dual Issue				
FPU Configuration	Interval of 1 instr.	Interval of 2 instr.s	Interval of 3 instr.s	Interval of 1 instr.	Interval of 2 instr.s	Interval of 3 instr.s	Interval of 4 instr.s	Memory System
#1 (Aurora III)	0.7	0.6	0.5	0.9	0.9	0.7	0.7	3 Cycle Latency
#2 (Fast Transfer)	1.3	1.0	1.0	1.4	1.4	1.3	1.3	
#1 (Aurora III)	0.3	0.3	0.3	0.4	0.4	0.3	0.3	1 Cycle Latency
#2 (Fast Transfer)	1.0	0.8	0.7	1.2	1.2	1.0	1.0	

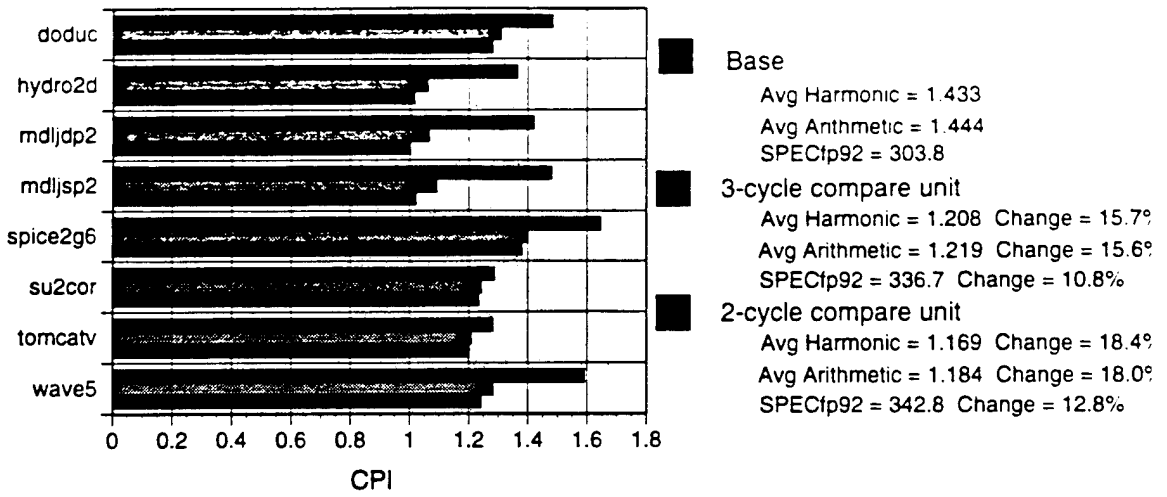


Figure 3.11 Upper Bound on Performance via Improved Compare Latency

involved in transferring the load data and compare instruction to the FPU. Figure 3.11 shows the results if compare latency is reduced from the 8+ cycles discussed above to either the two or three cycles that correspond to the latency of the add unit. The faster add unit offers only a 2% improvement in performance, a result which will be discussed further in Section 3.8.1. Several factors end to reduce the benefits of improved compare latency:

1) Not all compare instructions are preceded by a load. These instructions may suffer similar or worse latencies due to data dependencies. However, the data of Table 3.19 suggest that these occurrences are fairly infrequent.

2) Data cache misses will increase the latency of compare instructions beyond that assumed for this upper bound. Most SPEC benchmarks have relatively small data sets and experience good cache hit rates; "spice2g6" is a notable exception.

3) A compiler will not be able to schedule an arbitrary number of useful instructions between the branch, compare, and load instructions. As discussed above, smaller intervals between these instructions will increase branch-on-FPU stall cycles.

4) Some latency will always be associated with the compare instruction entering and exiting the reorder buffer. Support for precise memory exceptions requires that the compare be retired only when it reaches the head of the reorder buffer. This additional latency is on the order of 1.5 cycles.

Which components of compare latency are due to the inner workings of the FPU? Table 3.18 shows that in several benchmarks the result of a compare spends time in the reorder buffer before being committed to the status register. This is caused by other entries that precede the compare having not yet received a results from the functional units. This delay can add one or more cycles to the overall latency of a compare, raising the question of whether it is necessary to wait until a compare reaches the head of the reorder buffer before the compare is retired. Consider the following sequence:

```
1. c.eq          <= a floating-point compare
2. lw
3. bc1x          <= branch on compare being true/false
4. c.eq
5. nop
```

The first compare (1) completes, and upon exiting the floating-point add unit, writes the status register with a condition equal to one. The branch is then taken, followed by the second compare which evaluates to zero. At this point, the MMU determines that the load (2) causes a page fault. The integer reorder buffer entry corresponding to the load may have reached the head of the integer reorder buffer, but will not be committed to the status register due to the exception. All instructions which follow the load are squashed, the page fault is serviced and execution resumes at the load. However, this is the second time the branch is encountered and the condition in the floating-point status register is now zero, not one. Consequently, the wrong path is selected for the branch. On the other hand, waiting until the compare reaches the head of the floating-point reorder buffer will ensure that this error doesn't occur, since there will also be an entry in this reorder buffer for the integer load (see Section 4.5). As in the integer reorder buffer, this load entry will reach the head of the floating-point reorder buffer and stall until the IPU sends the FPU a signal indicating that the load cannot cause an exception. Though it is impractical to remove reorder buffer latencies from compares, as has been shown, it is interesting to see how much effect removing this latency might have on overall performance. An estimate for the upper bound on

Table 3.22 Removing Reorder Buffer Latency for Compares

Benchmark	Baseline (CPI)	No ROB Latency (CPI)
doduc	1.484	1.446
hydro2d	1.020	1.293
mdljdp2	1.420	1.356
mdljsp2	1.480	1.410
spice2g6	1.646	1.625
su2cor	1.286	1.277
tomcatv	1.281	1.273
wave5	1.589	1.538
Avg Harmonic	1.372	1.383
Avg Arithmetic	1.401	1.392
SPECfp92	303.8	308.9
% Change from Baseline		1.7

savings is:

$$\frac{cycles - (\frac{bc1x \text{ instructions that stall}}{Number\ bc1x \text{ instructions}} \cdot Latency_{rob})}{instructions} = CPI_{new}$$

On average across the benchmarks that experience branch-on-FPU stalls, the life-time for a compare in the reorder buffer is 1.18 cycles. Table 3.22 shows that this reorder buffer latency has a very minimal effect on overall performance.

3.7 Memory System Issues

This section focuses on improvements for the memory system to better support the execution of floating-point code, including the bandwidth to the primary data cache, the use of prefetching to minimize the absence of an on-chip data cache, the use of split integer and floating-point caches, and the organization of the IPU-FPU interface.

3.7.1 Double-word Loads and Stores

Currently, most applications that utilize floating-point numbers use the double-precision format which requires two loads or stores per operand. Since loads occur with a 25% frequency and stores happen 9% of the time, a wider path to the memory system would seem to offer significant performance gains. However, since both words of most double-precision operands hit in the data cache, the pipelined memory system of the Aurora III architecture means that the second reference delays issue by only one cycle (perhaps a bit more due to bus contention). In the somewhat unlikely event that the operands are in separate cache lines, a cache miss for the second word could impose many more than just one additional cycle of latency. Assuming the former case, an upper bound for the performance gained by supporting double-word references is:

$$CPI_{new} = \frac{-(\text{number of single word loads})/2 + cycles_{total}}{instructions_{total}}$$

This relationship assumes that load latency directly equates to IPU-FPU synchronization stalls, which for an upper bound is not an unreasonable assumption since we've seen that floating-point branches often wait for compares which are dependent on data from a floating-point load. If the number of loads is reduced to match the number of floating-point branches that stall while waiting for a compare, a new relationship is:

$$CPI_{new} = \frac{-(\text{floating-point branches that stall}) + cycles_{total}}{instructions_{total}}$$

In practice, the majority of comparisons are generated by a few sections of code. The actual performance benefit for double-word loads may lie between these two relationships. Figure 3.12 gives these bounds for all of the benchmarks. The benefit of double-word stores will be low because they occur infrequently; it is also rare to see synchronization stalls due to an inability to evict floating-point entries in the write cache (refer to Table 3.15). To confirm these estimates, the benchmarks should be recompiled to take advantage of double-word loads and stores. This will require the use of a newer version of Pixie, which is unavailable at the present time.

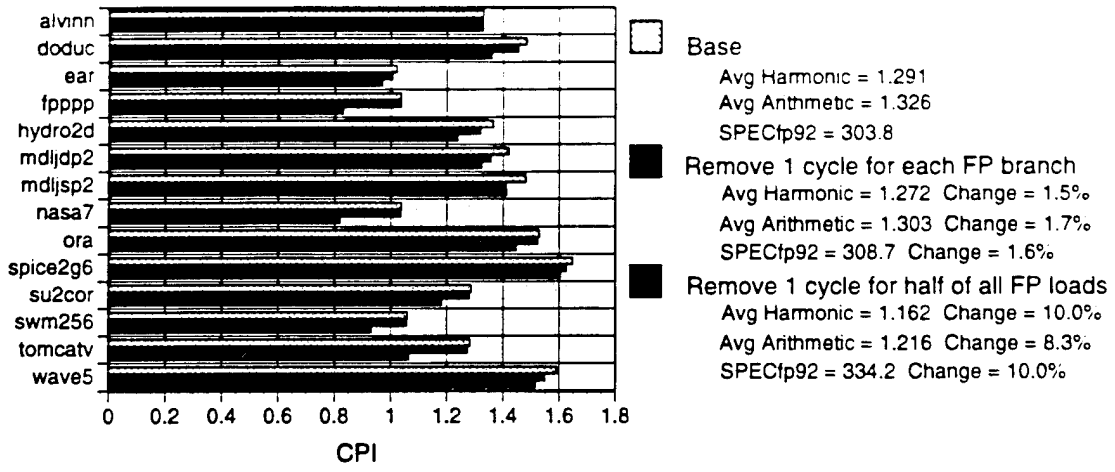


Figure 3.12 Performance Improvement via Double-Word Load Instructions

3.7.2 Prefetching of Data

Prefetching floating-point data serves to offset not having a large on-chip primary data cache and can result in a significant increase in performance. The overall improvement in CPI is 14.4% and some applications see a gain of as much as 60%. These results are based on a scheme that prefetches only with a unity stride, and examines only the top entry of the prefetch buffer; both of these constraints might warrant reexamination. Much work has been done elsewhere concerning hardware and software prefetching and can be referenced for these issues [Chen94], [Fu92], [Klaiber91], [Jouppi90].

3.7.3 Improving Cache Performance for Floating-Point Code

Since floating-point data access frequency can be 20-30% of total accesses, floating-point memory references can contaminate integer data in the data cache. Integer and floating-point data may be located in the same cache line, and a floating-point miss may flush needed integer data. The use of a split data cache would avoid this problem, but would require much additional overhead circuitry. The FPU would need logic to generate addresses and implement a cache coherency policy. An internal report from Princeton [Wolfe92] explores this idea in a study based on a modified version of Mike Johnson's Match simu-

lator. The integer processor contained parallel execution units for loads, stores, branches, and ALU operations. The FPU contained units for loads, stores, adds, conversions, multiplies, and divides. Reservation stations of 4 entries were used on all units except for division, which had 2 entries. The I-cache was 16k, 4 words/line, direct-mapped, with a 12-cycle miss penalty. The base system had a unified data cache of 64K, having 8-way associativity, 8 words/line, and a 16-cycle miss penalty. The split approach used a 16K integer data cache and a 64K floating-point data cache. A snoopy coherence policy was used for main memory; tags for both data caches were compared simultaneously in order to resolve store hits. The instruction fetch width, and corresponding number of decode units, was found to be four (the peak parallelism for instruction issue was five). Eight of the ten SPEC89 benchmarks were used (espresso and gcc were not used, due to disk constraints) and no more than 8 million instructions (without sampling) were run for any one program (possibly limiting the accuracy of the results). Floating-point performance in this simulator was almost identical performance for the split cache and the 64K unified cache. Integer performance dropped by 25-30% because of the reduced size of the integer data cache. In addition, the amount of invalidation between the two caches (a measure of how physical locality of integer and floating-point data) was only a few percent. Even if these invalidates were totally eliminated, the overall gain would be minimal. Similar performance could be obtained by simply increasing the size of the unified cache, without the extra hardware expense involved with splitting the caches.

A second idea for reducing cache contention between floating-point and integer data is not caching selected floating-point references. Floating-point intensive programs often stream through matrix/vector data; if the lifetime of this data is short, caching it may not make sense. Dynamic stride-detection and data prefetching could be used to recognize these cases and reduce their latency. This approach is more practical for a technology like GaAs which cannot support large on-chip caches because it reduces the occurrence of data being purged from the first-level cache. However, the Princeton split-cache study suggests that contention between integer and floating-point data may actually occur infrequently. This is another case where good compiler support could improve performance by organiz-

ing matrix data to increase its lifetime.

3.7.4 Interface between IPU and FPU

As mentioned before, the existing external IPU busses could be used in several different ways to handle the transfer of floating-point instructions and data. The simplest, and the one implemented for the FPU, uses only the two data cache busses, dcIn and dcOut (refer to Figure 3.1). The dcIn bus is used by the load-store unit to send data to the primary data cache and the dcOut bus is used to receive data from the data cache. From the perspective of the IPU, these busses are unidirectional. However, from the FPU perspective, both are bidirectional: instructions and data can be sent to the FPU via dcIn, and the FPU can send data to the data cache via dcIn; the FPU can receive data from the data cache via dcOut, and the FPU can send data to the IPU also via dcOut. Recall that there are 3 types of floating-point queues: instruction, load data, and store data. For a load instruction to be transferred, there must be a free entry in both the instruction and load queues. Each entry in the load queue has a valid bit, which indicates whether the data in the entry is ready to be used; the data comes from a valid write cache or data cache hit, or has been returned from the secondary memory system via the MMU and BIU. Instructions that transfer data from the IPU register file to the FPU register file will set this valid bit during the same cycle that the instruction and data are transferred. On the other hand, load data will be transferred after the load instruction, and then in a later cycle, when the status of the memory reference is known the valid bit will be set. Issue of a floating-point load which reaches the head of the instruction queue will be stalled until the valid bit for the head entry of the load queue is set. This approach might be modified to issue the instruction prior to knowing whether the data is valid, but the complexity in doing so is significant and simulations suggest the performance benefit is small. Since the data cache busses have multiple uses, a priority policy for each bus needs to be defined. The following summarizes the types of transfers that can appear on each bus.

dcIn

1) 2 floating-point instructions (64b total)

2) 1 double word store data operand (64b). For simplicity, single word loads/ stores will also be sent as 64 bits, with the upper half padded with zeroes. A pin on the FPU will indicate whether at least one store queue entry has valid data. Store data must be sent simultaneously to the IPU (in order to write the proper write-cache entry) and to the data cache. In other words, for a store transfer to occur, both dcIn and dcOut must be available. Decoupling this into 2 transfers would add complexity, which is probably not warranted, since stores infrequently (9% on average).

3) 1 move-to-FPU instruction (32b) and 1 operand (32b) from the integer register file. As mentioned above, the valid bit for the corresponding load queue entry will be set immediately upon transfer of the instruction and data.

4) 1 single/double word load data operand (64b). The majority of loads (80 to 95%) hit in the data cache, and therefore will be sent to the FPU via dcOut. However, some loads hit in the write cache or prefetch buffer, or need to come from the secondary memory system. In these cases, the data are transferred over the dcIn bus. The frequency of these types of transfers is quite low, on the order of a few percent of all cycles.

dcOut

1) 1 single/double word load operand (64b) on a data cache hit.

2) 1 move-from-FPU operand (32b) taken from the floating-point register file. As with a store, this transfer occurs via the store queue.

3) 1 single/double word store operand (64b).

Among these bus events, the transfer of load data is given the highest priority, since an unresolved load in the queue will block other instructions from issuing and can eventually stall instruction transfer if the queue becomes full. A deadlock situation might result if

instructions were given a higher priority than load data transfers. The only uses for the dcOut bus are for load data (integer and floating-point) and for store data (floating-point). As mentioned, stores are infrequent but loads are common. Loads via dcIn occur infrequently, but when they do occur, there is a good chance that the instruction queue has become full and has stalled the IPU. Therefore, the following priority policy was defined:

dcIn	dcOut
1. 1 single/double word load operand	1. 1 single/double word load operand
2. 1 or 2 floating-point instructions, or 1 move-to-FPU instruction with data	2. 1 move-from-FPU operand, or 1 single/double word store operand
3. 1 single/double word store operand	3. no transfer
4. no transfer	

The architectural simulator reports a number of interesting statistics about the utilization of these busses. As summarized in Table 3.23 and Table 3.24, bus activity varies greatly between the benchmarks; the busses are idle for some programs ("alvinn" and "spice2g6") whereas they are continuously busy for other benchmarks ("fpppp" and "nasa7"). Floating-point instructions comprise the great majority of transfers over the dcIn bus; they rarely stall due to a load-data transfer. This is to be expected, since load data is transferred via dcIn only on write-cache hits and data cache misses, both of which occur infrequently. Conversely, load-data transfers associated with data cache hits dominate the activity on the dcOut bus. For many benchmarks, store data transfers occur infrequently enough that they seldom contend for the use of either bus. An 8-entry write cache was used in these simulations and even those programs with a large number of store stalls due to bus conflicts experience very few IPU stalls. A more realistic 4-entry write cache still has few IPU stalls. Store stalls that result from bus contention impact performance only when the store queue is not large enough to accommodate store data while it waits for access to the

Table 3.23 Utilization of dcIn Bus

Benchmark	% Idle (of total cycles)	% FP Instrs (of total transfers)	% Loads (of total transfers)	% Stores (of total transfers)	% FP Instr Stalls (of total cycles)	% FP Store Stalls (of total cycles)
alvinn	99.2	77.6	0	22.4	0.000	0.000
doduc	54.2	76.6	0.2	23.2	0.013	8.834
ear	48.3	84.7	0	15.3	0.002	35.535
fpppp	9.7	72.7	0.1	27.2	0.008	45.802
hydro2d	50.3	76.7	3	20.3	0.282	16.678
mdljdp2	50	74.7	0.2	25.1	0.022	6.086
mdljsp2	62.5	80.2	0.1	19.7	0.010	3.430
nasa7	10.4	77.9	2.1	20	0.669	26.865
ora	47.9	64.3	0	35.7	0.000	18.056
spice2g6	92.1	88.8	8	3.2	0.002	0.142
su2cor	37.1	67.7	2.6	29.7	0.861	25.869
swm256	17.9	78.4	1.8	19.8	0.517	39.727
tomcatv	12.8	69.8	4.7	25.5	1.272	44.474
wave5	65.8	81.9	0.1	18	0.000	2.285

cache busses.

The Aurora III FPU was designed to use the existing data cache busses to support the transfer of floating-point instructions and data in order to minimize the number of pins on the IPU. This approach need not create a performance bottleneck, although it does complicate the design by requiring the prioritization of bus activity to be merged into existing load-store unit control logic. This decision requires the interface logic in the FPU to consider the various uses of the two cache busses and to ensure that instructions and data are sent to the appropriate registers. An alternative discussed earlier involves the use of dedicated busses between the IPU and FPU for transferring instructions and data. This would allow the transfer point for floating-point instructions to be moved ahead by several pipe stages, but would require adding approximately 192 pins to the IPU. The benefits of dedicated busses depend on each application and on the degree of synchronization between the

Table 3.24 Utilization of dcOut Bus

Benchmark	% Idle (of total cycles)	% Loads (of total transfers)	% Stores (of total transfers)	% FP Store Stalls (of total cycles)	Write Cache Eviction Stalls (4-entry)	Write Cache Eviction Stalls (8-entry)
alvinn	88.2	99.3	0.7	0.000	0.00	0.00
doduc	77.1	81.6	18.4	4.044	2.10	0.00
ear	83.8	93.5	6.5	8.829	0.10	0.00
fpppp	50.8	82.8	17.2	26.785	10.90	0.91
hydro2d	78.2	84.6	15.4	8.361	1.20	0.03
mdljdp2	81	74.4	25.6	2.097	6.40	1.15
mdljsp2	86.6	73.7	26.3	1.196	3.90	0.08
nasa7	47.7	84.2	15.8	17.426	0.00	0.00
ora	80.7	60.4	39.6	7.247	1.80	0.00
spice2g6	87.4	99.5	0.5	0.016	0.00	0.00
su2cor	75.3	69.8	30.2	9.403	12.0	0.91
swm256	67.9	80.4	19.6	14.513	19.60	0.58
tomcatv	53.6	78	22	26.230	7.00	0.00
wave5	86.2	80	20	0.644	0.00	0.00

IPU and FPU. Instruction sequences involving floating-point comparison and branch instructions have been found to have the greatest impact on performance. In order to benefit from transferring floating-point instructions earlier, both operands must be ready when the compare instruction reaches the issue point. The 3-cycle latency for a data cache hit in the Aurora III architecture would require more than 3 instructions to fall between load and compare instructions in order to take advantage of an early transfer; a shorter cache latency would favor an earlier transfer point, while dual issue reduces the number of cycles for intervening instructions, decreasing the impact of early floating-point instruction transfer.

3.8 Resource Allocation Issues

There are a number of issues that have an impact on the cost of allocating on-chip

resources, including the complexity of algorithms used for the various functional units, the use of pipelining to support higher clock frequencies while maintaining throughput, and the selection of a reasonable number of entries for the reorder buffer and queues in the FPU. RBE's are used to contrast the performance benefit with the cost of these resource alternatives.

3.8.1 Sensitivity to Functional Unit Latencies and Pipelining

In this section, the area and complexity costs of reducing latencies will be discussed for each of the functional units, and then the effect on overall performance of this reduced latency will be evaluated.

Numerous floating-point addition optimizations can be used to reduce latency, all at the expense of transistor count and implementation complexity. These include parallel paths for alignment and normalization, fast generation of the sticky bit, and leading one prediction for normalization. To evaluate the cost/performance ratio for these features, several adders were investigated. A two-cycle add unit incorporating these approaches occupied the most area, due primarily to its use of two 53-bit mantissa adders. A three cycle add unit resulted in an area reduction of 20%. Further reduction of resources of adder resources resulted in four- to five-cycle latencies.

Conventional approaches to multiplication involve a partial product array (3-2 or 4-2 carry-save adders) followed by a carry-propagate mantissa adder. Booth recoding of the input operands can be used to reduce the number of levels in the array by one, at the expense of adding recoding multiplexors. Analysis of these two approaches for GaAs DCFL showed that the area savings of Booth-recoding are small when compared to the increase in complexity of the design. Increases in capacitance along critical paths of a Booth recoded multiplier tend to offset the reduction in logic depth. Another alternative involves the iterative use of a smaller array. This approach reduces the size of the multiplier considerably, however five cycles are needed to produce a result. Furthermore, the multiplier is not pipelined, forcing subsequent multiply instructions to wait for the current instruction to com-

plete.

Non-restoring division algorithms can be enhanced by representing intermediate results in a higher radix redundant form. SRT-2, SRT-4, and SRT-8 approaches return one, two, and three bits per cycle, respectively. Latencies for these divide units vary from 20 to more than 50 cycles. Techniques can be employed for performing both on-the-fly conversion from redundant to sign-magnitude form and on-the-fly rounding of the result.

The IEEE floating-point standard specifies that conversion to and from all formats should be possible. For single, double, and integer formats, this results in six different types of conversion. While conversions could also be performed in the add unit, doing so would impact critical paths since the additional muxes and control required would result in an increase in logic depth. A separate conversion unit can be designed with a modest amount of hardware (30K transistors), having latencies of 2 to 4 cycles.

Figure 3.13 shows CPI performance for various latencies of the four execution units. Addition and multiplication both show a 17% change in CPI for latencies ranging from one to five cycles. For addition, latencies of two, three, and four correspond to interesting realistic designs. An add unit with a latency of two results in only a 2% improvement in performance over one taking three cycles, while a latency of 4 is 5% worse than a latency of 3. Similarly, each additional cycle of latency in the multiplier reduces performance (as measured by CPI) by 4%. In the division unit, over a latency range of 10 to 30 cycles, performance changes by 8%. Conversion instructions occur very infrequently and have little impact on overall performance. The same simulations were repeated for non-pipelined addition and multiplication units. Interestingly, the degradation in performance is less than 5%. The latches used for pipelining these two units account for approximately 25% of the area for each unit. Not pipelining these units would result in significant savings in area and power dissipation

3.8.2 Reorder Buffer

A reorder buffer provides many benefits. First, if an out-of-order completion policy

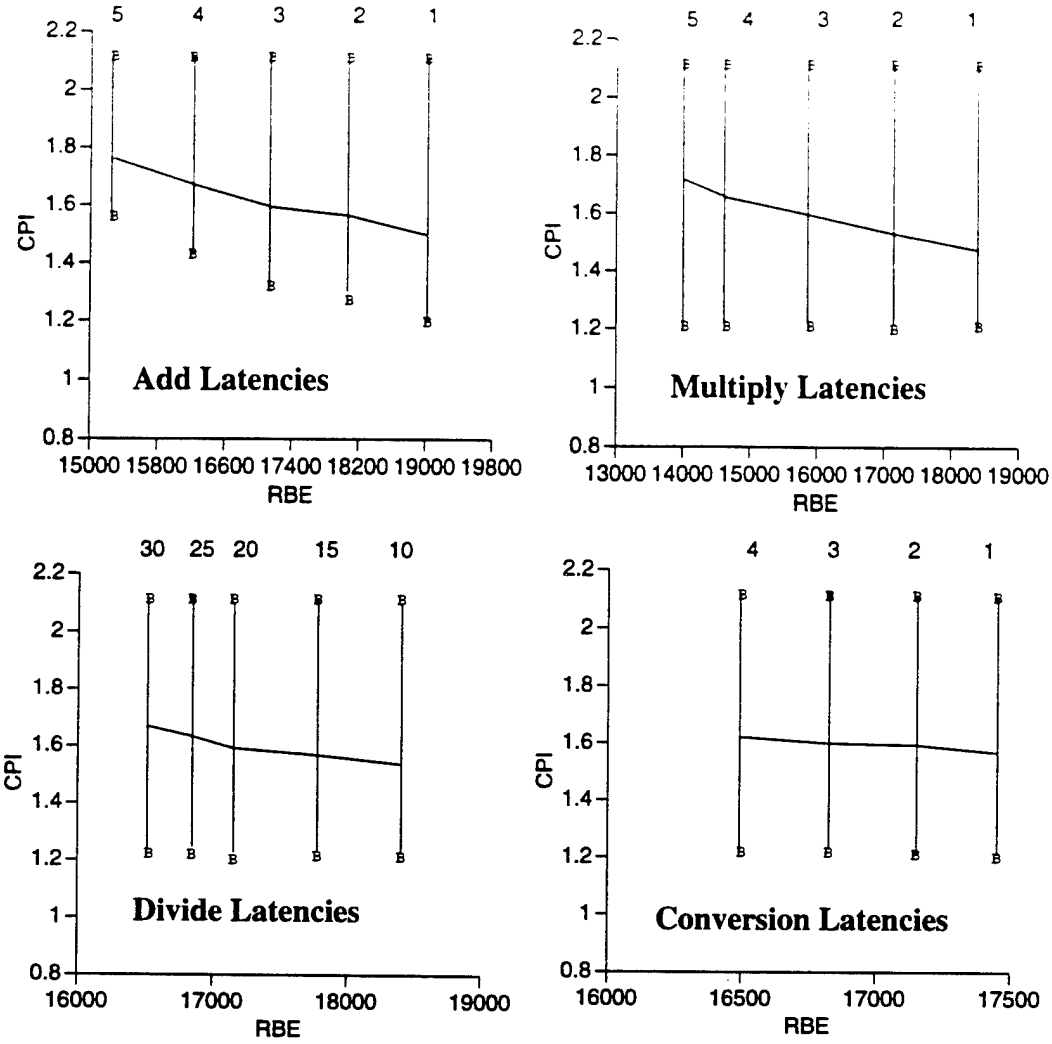


Figure 3.13 Performance vs. Resource Cost for Floating-Point Functional

is used, the reorder buffer ensures that results are written back to the register file in the correct order. Upon issue, an instruction reserves the next available location in the reorder buffer by writing the result register to the entry and by adding the reorder buffer tag to the correct result-bus shift register entry. When an instruction in an execution unit has finished, this tag is used to guide the result into the correct reorder buffer entry and the valid bit for the entry is set. On every clock cycle, if valid data is at the head of the reorder buffer, it is written back to the register file. In this way, the reorder buffer serves to prevent output dependencies, since results are written back in the order of the original instruction stream. Both output and anti-dependencies result from aggressive compiler scheduling of a limited number of registers (hardware parallelism is limiting available instruction parallelism),

rather than from true dependencies. Register renaming can be used to reduce their impact. This can be done in the reorder buffer by identifying a result entry with the specific result register. When a source operand is needed, the reorder buffer and the register file are accessed in parallel, the former being addressed using the register name. If the result is found in the reorder buffer, it is forwarded as the correct operand. Multiple result references to the same register can be active at one time within the reorder buffer; hence it is necessary that the most one be returned. As an alternative to an associative reorder buffer, which is difficult to implement in GaAs, a small tag lookup table is used to find the most recent reorder buffer tag for a source register. The valid bit in the entry determines whether the corresponding data in the reorder buffer is available. Finally, the reorder buffer can be used to ensure that exception handling is precise. If an exception occurs during the execution of an instruction, an exception field in the reorder buffer entry will be written at the same time as the exceptional result. When this entry reaches the head of the reorder buffer, an exception is signalled and all remaining entries are marked as invalid. Consequently, no subsequent instructions have written the register file, and after handling the exception, the FPU can be restarted at the instruction which follows the exceptional one.

The average number of entries needed in the reorder buffer, as found through simulation, reflects the number of simultaneous floating-point instructions in execution on average. Another way to view the size of the reorder buffer needed is by comparing how the average number of entries relates to the average number of floating-point instructions per basic block, as shown in Table 3.25. Several observations can be made from this table. First, the average size of a basic block tends to be much larger for floating-point intensive applications than for integer ones; much more work is being done between branches for floating-point code. Second, Table 3.25 suggests some correlation between the number of floating-point instructions per basic block and the optimal size for the floating-point reorder buffer. This reflects the effect of cache misses, which tends to reduce the number of outstanding floating-point instructions. As discussed above, the type of issue policy can affect how many instructions are active at a time and how many reorder buffer entries are needed; an OOOO policy requires more entries than an IOOO policy (an IOIO policy does not use

Table 3.25 Basic Block Sizes for SPECfp92

Benchmarks (Complete Run)	Instructions per Basic Block	Floating-Point Instructions per Basic Block	Reorder Buffer Entries (IOOO Policy)	Reorder Buffer Entries (OOOO Policy)
alvinn	13.97	9.96	1.17	1.17
doduc	11.33	6.86	6.29	8.75
ear	9.48	6.00	1.95	14.35
fpppp	27.92	21.72	8.50	10.22
hydro2d	8.65	4.89	3.78	7.39
mdljdp2	11.19	6.71	6.60	11.24
mdljsp2	9.60	4.97	3.59	9.45
nasa7	28.35	21.74	17.45	17.97
ora	8.80	5.216	7.03	8.14
spice2g6	10.77	1.543	1.12	2.36
su2cor	17.97	12.73	8.27	11.46
swm256	38.97	31.53	7.87	16.17
tomcatv	27.22	24.08	9.20	14.41
wave5	15.64	9.97	2.69	5.38
Avg Harmonic	13.54	6.51	3.4	5.7
Avg Arithmetic	17.13	11.99	6.1	9.9

a reorder buffer at all). Consequently, an OOOO policy requires roughly the same number of entries as there are floating-point instructions per basic block. Figure 3.14 compares performance to resource cost.

3.8.3 Instruction Queue

Queues have been a component of numerous past designs. For instance, in a DAE machine the instruction stream is split into separate decoupled memory and execute streams that communicate via queues [Smith86, Smith87]. This approach offers several advantages, including an opportunity to issue more than one instruction per cycle and less sensitivity to memory access delays, since the instruction fetch stream is allowed to run

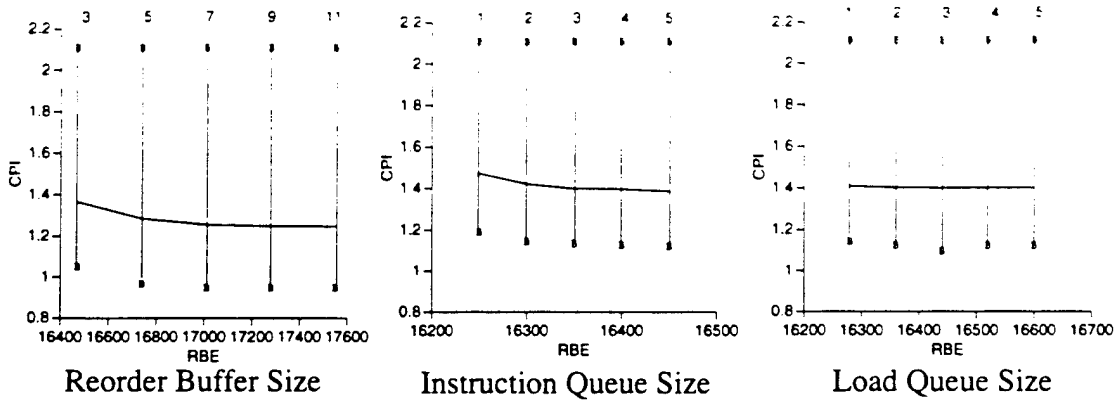


Figure 3.14 Queue and Reorder Buffer Resource Allocation

ahead of the execution stream. Thus, memory accesses can be dispatched further in advance of when they are actually used. For other designs, an I/O buffer has been a characteristic of at least one previous FPU [Takeda85], and instruction queues have appeared in several other machines [Molnar89], [Steiss91], [Darley90].

The use of an instruction queue instead of a single entry instruction buffer allows substantially more slip by the IPU. Instead of stalling due to FPU data dependencies or resource conflicts, the IPU is able to transfer a floating-point instruction and continue execution. The IPU will stall due to the FPU only when of the queue becomes full or when it has to wait for FPU results. Decoupling queues also serve to hide extra latency caused by chip-crossings, and can lessen the impact of having different clock frequencies for the IPU and FPU, a situation that can result from there being a difference between the bit-widths of datapaths for these two chips. This point is made in Table 3.26, that as the clock frequency of an IPU chip (paired with a 250MHz FPU) is increased, the CPI does not degrade as fast as the IPU frequency increases.

Figure 3.14 shows how performance varies for different queue sizes. For a dual issue policy, a queue of depth 5 achieves nearly the same performance across all benchmarks as a queue with an unlimited number of entries. The same experiment was performed for a single transfer, single issue processor; the results (not shown) indicate an optimal queue size of 3 entries. Considering that only one instruction can be transferred from the IPU to the FPU per cycle in such a system, and that there are fewer floating-point instructions ac-

Table 3.26 CPI for Queues and Different IPU/FPU Clock Frequencies

Benchmark (50M Instructions)	250MHz (IPU = FPU)	300MHz (IPU = 1.2 FPU)	350MHz (IPU = 1.4 FPU)	400MHz (IPU = 1.6 FPU)
ear	1.0049	1.1603	1.3170	1.4941
fpppp	1.0236	1.1172	1.2678	1.4257
hydro2d	1.2866	1.4364	1.5634	1.7056
spice2g6	1.6238	1.6740	1.7001	1.7318
Avg Harmonic	1.189	1.311 (10.3%)	1.441 (21.2%)	1.578 (32.7%)
Avg Arithmetic	1.235	1.347 (9.1%)	1.462 (18.4%)	1.589 (28.7%)

tive at a time in such a machine, the smaller optimal queue size is to be expected. Figure 3.15 shows how the recommended size of the instruction queue varies with issue policy. The smaller number of entries needed in the instruction queue for an OOOO policy can be explained by the fact that dispatch constraints are less restrictive; as a result, instructions spend less time on average in the queue before reaching a reservation station. In effect, look-ahead is being expanded with the larger instruction window. This decrease in queue entries is more than compensated for by a need to have more reorder buffer entries, which are more expensive. An upper limit for the number of entries in the instruction queue would be the number of entries in the integer reorder buffer, since there cannot be more than that number of active floating-point instructions. The use of an instruction queue does make

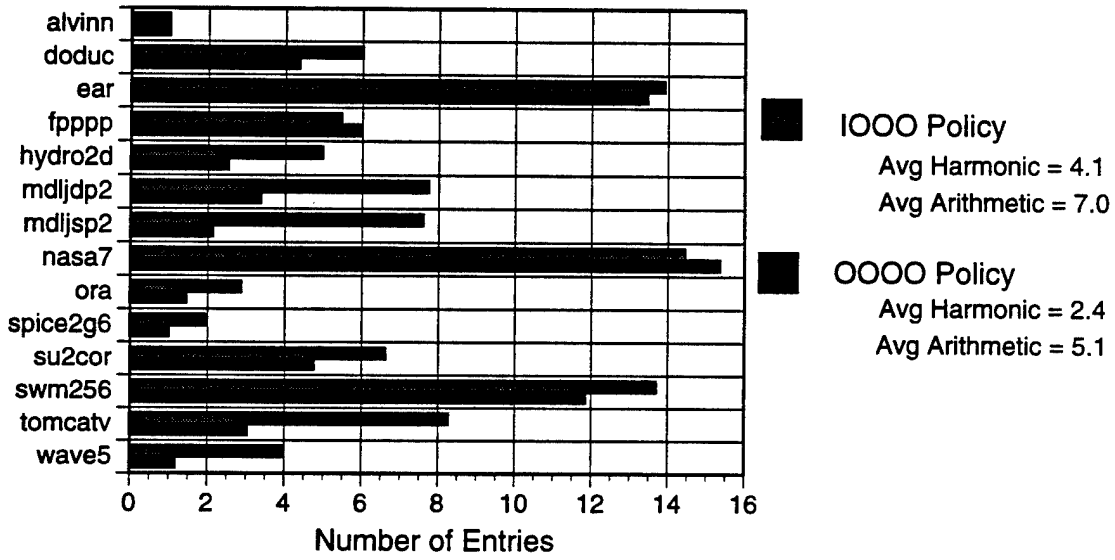


Figure 3.15 Instruction Queue Size

precise handling of exceptions more costly in terms of hardware, since the IPU may be many instructions past an exceptional instruction when the exception is signalled. However, as will be discussed later, there are several ways to address imprecise floating-point exceptions.

3.8.4 Load Queue

Use of an instruction queue necessitates a corresponding load queue in which to store data prior to its being written into the floating-point register file. However, the depth of this queue does not need to be as great as that of the instruction queue. As shown in Table 3.27, for an IOOO policy, a 3 or 4 entry load queue functions well, while an OOOO policy needs one fewer entry. As in the instruction queue, out-of-order issue moves more instructions past the queues and into reservation stations, which also explains the need for fewer entries in the load queue. The recommended size of the load queue correlates well with the fact that most floating-point operations use double precision operands, each of which requires 2 single-word load instructions. The need for an additional entry can be accounted for by memory system latency. Even on a cache hit, there is one cycle delay between arrival of the load instruction at the queue and validation of the data. Of course, additional delay is incurred on a cache miss; the average number of load queue entries in the table accounts for cache miss latencies, too.

3.9 Miscellaneous Issues

This section will address a number of unrelated issues, including hardware support for square root, several alternatives for implementing floating-point store instructions, the use of additional result busses to support an increase in parallelism for dual issue of floating-point instructions, the use of the multiply unit to perform division, and the trade-offs in implementing different floating-point operand widths.

3.9.1 Hardware Square Root

Since transcendental functions occur infrequently in the SPECfp92 benchmarks, they are not worth implementing in hardware. The Aurora III FPU omits these operations, depending instead on the compiler to link in software library routines to perform these calculations. However, the question of whether to support square root in hardware is not as clear; Figure 3.16 suggests that some programs would benefit from doing so. Square root capability with a latency of approximately 21 cycles can be built onto a divide unit; this requires the addition of both a lookup table ROM for deriving an initial estimate and some extra datapath logic. The impact on area is small; the impact on critical paths may be slightly greater. The performance of such square root hardware can be compared to the MIPS li-

Table 3.27 Load Queue Size

Benchmarks (50M Instructions)	Average Load Queue Entries (IOOO Policy)	Average Load Queue Entries (OOOO Policy)
alvinn	1.00	1.00
doduc	3.07	2.51
ear	3.60	3.59
fpppp	3.54	4.02
hydro2d	3.26	1.78
mdljdp2	3.17	1.68
mdljsp2	2.02	1.28
nasa7	8.07	8.38
ora	1.24	1.01
spice2g6	1.71	1.00
su2cor	3.09	2.14
swm256	4.80	4.31
tomcatv	4.36	1.79
wave5	1.88	1.03
Avg Harmonic	2.4	1.7
Avg Arithmetic	3.2	2.5

brary function for square root, which executes 62 instructions, with a CPI for the routine of 2.09, meaning that 124 cycles are needed for each square root reference. A bound on CPI that is possible for a hardware square root instruction is described by the following:

$$CPI_{new} = \frac{Cycles_{old} + (\text{Number sqrt references} \cdot Latency_{sqrt}) - (\text{Number sqrt instructions per reference} \cdot CPI_{sqrt})}{instructions}$$

$Latency_{sqrt}$ is the number of cycles added by a square root instruction to the total runtime. In the worst-case, the FPU will stall as soon as the square root instruction is issued, and the entire latency of the operation will be added to the total cycle count. Intermediate points in the design space might assume that useful integer work is overlapped with the square root operation and that the effective latency is either 0 or 10 cycles. Figure 3.17 summarizes the impact of square root for the 8 benchmarks which use this function. While the improvement for one application, "ora", is quite large, the average improvement is a more modest 7% to 9%. The "ora" benchmark represents a class of programs which perform graphics transformations and for which square root is used frequently. Since multi-media applications are increasing, other programs of this type should be examined in order to further illuminate the cost/benefit of including a hardware square root instruction. The rate of performance growth for processors suggests that any feature which adds only 5% to overall performance should require no more than a month for design and verification.

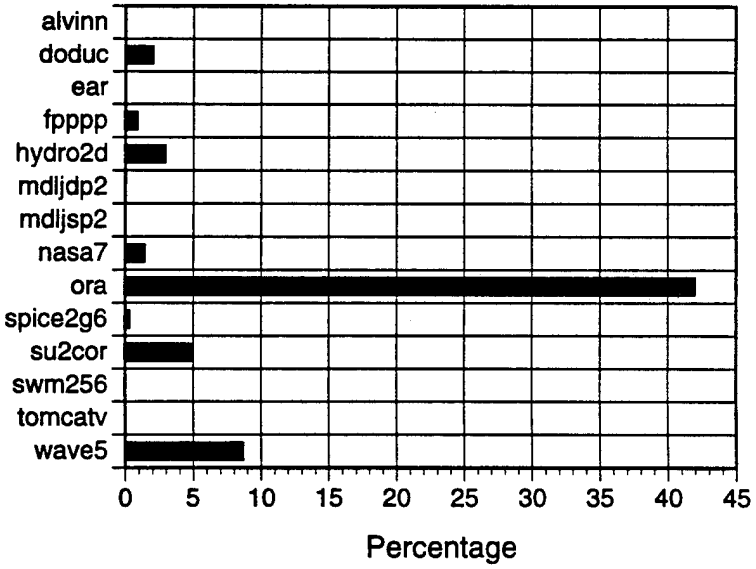


Figure 3.16 Percentage of Instructions Due to Square Root

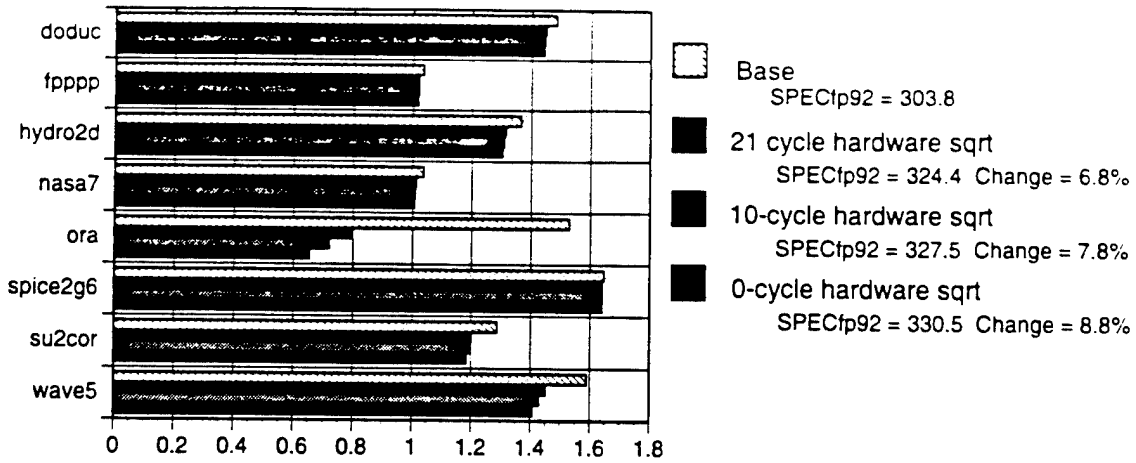


Figure 3.17 Performance Benefit via Hardware Support for Square Root

3.9.2 Store Policies

Data transfers from the FPU to the IPU can be handled in several ways. As with other floating-point instructions, store and move-from-FPU instructions (both will be referred to as store instructions) must return results in the correct order. One simple approach would be to stall the FPU when a store instruction reaches the head of the instruction queue if the corresponding source register has a write-back pending. However, this scheme would also stall instructions following the store instruction which would otherwise be able to issue. In this case, two kinds of out-of-order issue might make sense. The store instruction might be issued to the reorder buffer, where it would await completion of the instruction producing the data to be stored. In-order completion would be ensured by the reorder buffer; when the store entry reached the head of the reorder buffer and contained valid data, it would be sent to the IPU via a store queue. This manner of issuing store instructions requires one additional write port in the reorder buffer for each result bus, since a result might need to be written to both its own entry and to a store entry. More importantly, the store transfer would not take place until all entries ahead of it in the reorder buffer had been written back to the register file.

Another approach would involve the use of a separate store reorder buffer. A tag would be written to the store reorder buffer at issue and when the pertinent data was available, it would be written to both reorder buffers. An additional result shift register (de-

Table 3.28 Store Policies

Benchmarks (65M Instructions)	CPI Stall on Issue	CPI Issue to ROB	CPI Issue to Store ROB
alvinn	2.111	2.110	2.110
doduc	1.768	1.770	1.754
ear	1.136	1.091	1.091
hydro2d	1.099	1.093	1.081
mdljdp2	1.079	1.072	1.072
nasa7	1.065	1.034	1.032
ora	1.768	2.002	1.770
spice2g6	1.194	1.204	1.204
su2cor	1.683	1.712	1.656
Average	1.345	1.347	1.326
% Change		0.2	1.4

scribed under Section 3.9.3) would be needed, but a register lookup table would not be needed, since the result would be accessed for on-chip use only through the regular reorder buffer. The most important characteristic of the two-reorder buffer approach is that stores could be completed ahead of previous non-store entries that have not yet completed. The peak number of entries needed in such a store reorder buffer would relate to the largest possible number of active instructions, although the average number of entries would be much lower. As indicated in Table 3.28, stall-on-issue and issue-to-ROB policies would achieve similar performance; an issue-to-store ROB architecture would achieve only a 1.4% improvement in overall CPI compared to a stall-on-issue design (individual applications saw as much as a 4% improvement in CPI). The small performance gain for a store reorder buffer does not justify its use. Store instructions are used fairly infrequently and when they are encountered there is a high probability that the necessary operand has already been, or will soon be, computed, since the most commonly used floating-point instructions have fairly short latencies. Table 3.29 shows that little time expires between a floating-point store reaching the issue point and data being ready. Support for an appropriate number of write

Table 3.29 Average Delay Between Issue of a Store Issues and Data Availability

Benchmark (65M Instructions)	Average Store Reorder Buffer Wait for Data
doduc	1.621
ear	1.036
fpppp	1.357
hydro2d	1.483
nasa7	1.378
spice2g6	1.079
su2cor	1.468
swm256	1.443
tomcatv	1.492
Avg Harmonic	1.345
Avg Arithmetic	1.373

cache entries would further decouple the execution of floating-point stores. The store reorder buffer would also face problems handling memory exceptions and preserving a precise execution model.

As a result, the Aurora III FPU was implemented with a stall-on-issue policy for stores in conjunction with a simple store queue. To ensure precise memory exceptions, this queue is written with data that has passed through the reorder buffer. The store data originates from the floating-point register file, reorder buffer, status register, or HI/LO registers (used for integer multiplication). Each store queue entry contains: 1) the data operand, aligned to the high or low word, depending on the address of the store, or aligned to the low word for move-from-FPU instructions, 2) a 2-bit tag which identifies the type of store (move-from-FPU or store single/double), and 3) the integer reorder buffer tag for the store instruction, which is used to guide the data to the correct write-cache or integer reorder buffer entry. A single FPU pin is used to indicate that floating-point store data is available and ready to be transferred to the IPU. Table 3.30 summarizes the average number of store queue entries needed by each benchmark; a size of 2 or 3 entries is consistent with the ob-

Table 3.30 Average Number of Store Queue Entries

Benchmarks (50M Instructions)	Avg Store Queue Entries (IOOO Policy)
alvinn	1.000
doduc	3.442
ear	1.108
fpppp	5.951
hydro2d	3.001
mdljdp2	3.176
mdljsp2	1.873
nasa7	3.000
ora	1.933
spice2g6	1.433
su2cor	5.859
swm256	2.950
tomcatv	5.311
wave5	1.303
Avg Harmonic	2.135
Avg Arithmetic	2.953

servation that most floating-point stores are used with double-precision operands.

3.9.3 Result Busses

As execution of instructions increases within the FPU, more demand is placed on the bus used to write results from an execution unit to the reorder buffer. Consequently, having additional result busses is beneficial, especially in supporting dual issue of floating-point instructions. From Table 3.31, we can conclude that no more than two result busses are needed, which is consistent with results presented elsewhere for designs which issue on average between one and two instructions per cycle [Johnson91]. Each result bus has a corresponding result shift register (RSR), each entry of which contains a reorder buffer tag and a functional unit designator. The output of the functional unit indicated by the designator is

Table 3.31 CPI Performance for Different Numbers of Result Busses

Benchmark (65M Instructions)	One Bus (CPI)	Two Busses (CPI)	Three Busses (CPI)	Four Busses (CPI)
alvinn	2.111	2.111	2.111	2.111
doduc	1.704	1.688	1.696	1.696
ear	1.146	1.127	1.127	1.127
hydro2d	1.052	1.035	1.035	1.035
mdljdp2	1.086	1.019	1.019	1.019
nasa7	1.279	1.057	1.057	1.057
ora	1.775	1.764	1.764	1.764
spice2g6	1.204	1.204	1.204	1.204
su2cor	1.654	1.627	1.627	1.627
Average	1.367	1.312	1.312	1.312
% Change		4.0	4.0	4.0

written to the appropriate reorder buffer entry, through the use of the tag. The RSR needs to have one stage of depth for each cycle of delay in the functional unit with the longest latency. By constraining divides to use only one of the result busses, one can reduce the number of entries in the RSR for the other result bus. Alternatively, the Aurora III FPU allows divide instructions to poll the result busses upon completion of a divide operation; this means that writeback of the reorder buffer is delayed until a free bus can be acquired. This approach saves stages for the result bus shift registers, at a slight cost in divide latency.

3.9.4 Divide Performed in Multiply Unit

There are numerous hardware algorithms for performing division, including restoring, non-restoring (SRT, Sweeney-Robertson-Tocher), and successive approximation (i.e., Newton-Raphson). The Newton-Raphson method, which involves multiplication by a reciprocal, could be done within the multiply unit, saving the area of a dedicated divide unit. The effects of doing so were simulated under the following assumptions:

1. a divide can be issued even if multiplies are currently being executed in the multiply

Table 3.32 Divide Performed in Multiply Unit

Benchmark (65M Instructions)	CPI
alvinn	2.127
doduc	2.038
ear	1.623
hydro2d	1.115
mdljdp2	1.800
nasa7	1.294
ora	1.921
spice2g6	1.204
su2cor	1.929
Avg Harmonic	1.587
Avg Arithmetic	1.672
% Change from IOOO baseline	-20.7

unit, since each will be using different stages of the multiply pipeline.

2. a divide or multiply cannot be issued to the multiply unit if a divide is currently being executed.

The results of this experiment, which uses an IOOO policy and a 3-entry instruction queue, are shown in Table 3.32; the use of a pipelined multiply unit to perform divides has a significantly negative impact on overall performance. In fact, the gains derived from using an IOOO policy and an instruction queue have been lost. As a result, a separate divide unit has been implemented; this unit has been omitted from the current FPU design due to space limitations. It would be interesting to determine the effect of not doing division in hardware at all, but instead in software. While most applications avoid division (less than 1% of all SPECfp92 instructions are floating-point divides), certain programs, such as graphics applications, would suffer a significant performance loss without a hardware division instruction.

3.9.5 Operand Formats

To minimize design time of the FPU, single precision calculations were not included in the add, multiply, or divide units; conversions among all data formats are supported by the conversion unit. The effect of this decision on overall performance is minimized by the fact that the ANSI standard for C requires all floating-point calculations to be performed using the double precision format. Consequently, for the two single-precision SPECfp92 benchmarks that are compiled in C, the compiler converts all single precision operands into the double precision format prior to performing a floating-point arithmetic operation. Before a result is stored to memory, it is converted back to the single precision format. All of these conversions add overhead to the execution of the program. In "alvinn", conversion instructions account for 1.0% of all instructions and for "ear", conversions comprise 16.9% of all floating-point instructions. There are several options for managing this problem: 1) change all program variables to the double precision format, and 2) support single precision operands in hardware. In either case, an upper bound on performance improvement can be derived from the following relationship:

$$Cycles_{new} = Cycles_{old} - (\text{Number conversion instructions} \cdot CPI_{Convert})$$

This expression assumes that the majority of conversion instructions are used to convert between the single and double formats, and not between the word format; this assumption is valid for the "alvinn" and "ear" benchmarks. The $CPI_{convert}$ variable represents the average number of cycles needed to execute a conversion instruction; several realistic values are used in Table 3.33. The total cycles for the "alvinn" benchmark is not affected since conversions comprise such a small percentage of the total instructions that are executed. On the other hand, for "ear" it is clear that a fairly significant improvement in performance can be realized by eliminating the need to perform these conversions. As a reference point, if the performance of a single benchmark improves by 25%, the overall SPEC rating will improve by 1.3%. The other 4 benchmarks that use single precision operands (mdljsp2, swm256, tomcatv, wave5) are all compiled in Fortran, which does not coerce data into a double precision representation. As a result, the CPI values for these

Table 3.33 Overhead for Conversions in Single Precision Benchmarks

Benchmark	Cycles for $CPI_{convert}$ = 1.0 and % Reduction	Cycles for $CPI_{convert}$ = 1.25 and % reduction	Cycles for $CPI_{convert}$ = 1.5 and % reduction
alvinn	66.35M 0.00%	66.35M 0.00%	66.35M 0.0%
ear	41.93M 16.5%	39.68M 20.6%	37.83M 24.7%

benchmarks already assume that one of the two approaches just described has been employed in order to avoid generating additional conversion instructions. Conversions are not significant for any of these Fortran programs, accounting for less than a few percent of all instructions. Across all 5 single-precision benchmarks, these alternatives for handling single precision numbers can improve performance by no more than about 5%. It is interesting to note that the Multi-Titan FPU [Jouppi88] operates only on double precision numbers and that the RS/6000 internally converts from single to double operands prior to performing an operation.

The IEEE 754 specification also recommends that the extended format be included for the widest basic format. This leads to the following combinations: single and extended single, double and extended double, single and double and extended double. However, in order to follow this and support the double precision format, it would be necessary to have a datapath width of more than 80 bits for the extended format. It is possible that having an extended format might be used to reduce exceptions in sensitive pieces of code. However, an extended double format is just too costly in terms of hardware for the integration constraints of GaAs, and is probably not justified considering how infrequently it would be used.

3.10 Simulation Accuracy Issues

Although the Aurora III design implements static branch prediction, the simulator does not model this feature, instead assuming perfect prediction. The actual performance of the processor will be lower than what has been presented so far, and an estimate for the impact of miss-predicted branches is presented. The combination of simulation speed and

size of the design space to be explored constrain the runtime of any given experiment. The effect of simulating only the first 50 million instructions is examined and sampling is discussed as a means of improving simulation speed and accuracy.

3.10.1 Branch Prediction

The Aurora III architectural simulator does not model either a static or dynamic branch prediction policy. Consequently, all branches are assumed to be predicted correctly and no effort is made to simulate the effect on CPI of recovery due to miss-prediction. However, the measured values for CPI can be derated by using a simple relationship:

$$CPI_{new} = CPI_{old} + \left(CPI_{old} \cdot \frac{\text{branch instructions}}{instructions} \cdot \frac{\text{miss predicted branches}}{\text{branch instructions}} \cdot \text{cycles needed to restart}\right)$$

Table 3.34 summarizes the penalty for several choices of prediction rate and pipeline depth, assuming an initial CPI of 1.3. The Aurora III design has a pipeline depth such that restarting issue after a miss-predicted branch will require squashing 3 cycles of instruc-

Table 3.34 Effect of Branch Prediction on CPI = 1.3 (New CPI and %Change)

Prediction Accuracy for Integer (25% branches) and FP (5% branches)	2 Cycle Miss Predict Penalty CPI % Change	3 Cycle Miss Predict Penalty CPI % Change	4 Cycle Miss Predict Penalty CPI % Change	5 Cycle Miss Predict Penalty CPI % Change
2-Level History Table (0.96)	1.326 2.0	1.339 3.0	1.352 4.0	1.365 5.0
2-Bit Counter (0.89)	1.372 5.5	1.407 8.3	1.443 11.0	1.479 13.8
1-Bit Counter (0.83)	1.411 8.5	1.466 12.8	1.521 17.0	1.576 21.3
BTFN (0.61)	1.554 19.5	1.680 29.3	1.807 39.0	1.934 48.8
2-Level History Table (0.98)	1.303 0.2	1.304 0.3	1.305 0.4	1.306 0.5
2-Bit Counter (0.96)	1.305 0.4	1.308 0.6	1.310 0.8	1.313 1.0
1-Bit Counter (0.94)	1.308 0.6	1.312 0.9	1.316 1.2	1.320 1.5
BTFN (0.71)	1.338 2.9	1.357 4.4	1.375 5.8	1.394 7.3

tions that are currently in execution. Branches for integer code occur 25% of the time and 5% for floating-point programs [Yeh93]; this is related to the observation that basic block size is much larger for floating-point code than for integer code. There are numerous prediction policies, including: 1) two-level history table, 2) 2-bit history counter, 3) software profiling (static), 4) 1-bit counter, 5) backward-taken, forward-not-taken (static), 6) always taken (static). Table 3.34 focuses on the first, second, fourth, and fifth of these policies, since these represent a reasonable range of possible prediction rates. Clearly, a more deeply pipelined machine needs to have a better (and possibly more costly) prediction policy. However, even for the worst policy, floating-point CPI increases by only 4.4% for the Aurora III FPU. Losses in CPI due to miss-predicted branches are offset to some degree by the positive effects that are not modeled in the simulator, such as support for both double-word loads/stores and 64 floating-point registers.

3.10.2 Sampling and Simulation Speed/Accuracy

With the design trade-offs all dependent on simulation results, one is naturally interested in the accuracy of trace-driven simulation and whether runtime can be reduced. The experiments presented depend on results that have been collected from simulating the first 50 to 100 million instructions for each benchmark. We need to know whether the beginning portion of a program is representative of the entire runtime, especially considering the possibility that these first instructions may comprise one-time initialization events? Short of executing and simulating the entire program, the accuracy of the results can be in question. Closely tied to this issue is the problem of constraining how long it takes to perform an experiment. The total runtime can grow rapidly as one considers different points within a design space across a broad set of benchmarks. Sampling is a technique originally proposed for cache simulations in order to address these concerns[Laha88, Liu93, Poursepanh94]. Extending this approach, instead of running continuously for a certain number of instructions the simulator would start sampling at different times, ending each time after some number of instructions. This sampled instruction trace might be read from a file, although this would increase network traffic over taking traces directly from program execu-

tion, which would be problematic if numerous machines are running simulations in parallel. Alternatively, the trace might still be generated on-the-fly and the simulator turned on/off at appropriate intervals. The main bottleneck for trace-driven simulation is the speed of the simulator itself, not how long it takes to execute the benchmark; for example, current machines are fast enough to run many SPEC benchmarks in only a few tens of seconds. In order to avoid inaccuracies that might result from cold-start cache effects, hit rates for various cache structures (instruction, data, tlb) are derived for each benchmark and are applied to the sampled instruction and data traces. The accuracy of sampling can be correlated to long instruction runs through the use of certain comparison metrics. These include:

1. The number of integer and floating-point instructions per basic block. This is a measure of the distance between branches.
2. The frequency of different instruction types.
3. Cycles per instruction (CPI).
4. Breakdown of stalls and average latencies for the commonly occurring instruction types.

Table 3.35, Table 3.36, and Table 3.37 evaluate the error involved in only running for the first 50 million instructions of 4 benchmarks by comparing these metrics against longer runs of 1 billion instructions. Three of the four benchmarks experience only very small differences in CPI and cache hit rates; "spice2g6", the exception, experiences both higher cache miss rates and more branch-on-FPU stalls.There is more variation among the 3 floating-point stall sources for the different benchmarks; however, error for floating-point branch stalls is fairly small for the application (hydro2d) which is most impacted by this type of stall. For the other benchmarks, these 3 stalls each occur less than 20% of the time and often do not directly affect CPI because useful integer activity is performed during the floating-point stall. Instruction frequency changes only slightly for "fpppp" and "hydro2d", but changes more for the other 2 benchmarks. The occurrence of load and conversion instructions increases for the "ear" benchmark on longer simulations, but this does not translate into a large difference in CPI. The "spice2g6" benchmark demonstrates that there are

Table 3.35 Comparison Metrics (50M / 1G Instrs. and % Difference)

Metric	ear	fpppp	hydro2d	spice2g6
FP Instructions per Basic Block	4.48 / 5.92 32.14%	27.47 / 24.67 -10.19%	4.80 / 4.82 0.42%	0.64 / 0.25 -60.94%
Instructions per Basic Block	9.17 / 9.47 3.27%	33.56 / 30.82 -8.16%	8.50 / 8.53 0.35%	5.59 / 7.98 42.75%
CPI	1.005 / 1.011 0.06%	1.024 / 1.020 -0.39%	1.287 / 1.332 3.49%	1.624 / 1.387 14.59%
I-cache Hit Rate	99.90 / 99.93 0.00%	78.00 / 78.71 0.91%	99.94 / 99.97 0.003%	97.57 / 99.28 1.75%
D-cache Hit Rate	99.14 / 99.92 0.79%	99.87 / 99.95 0.08%	92.47 / 92.49 0.02%	90.66 / 94.25 3.96%
Iq/Lq Full (% cycles)	9.97 / 14.52 31.34%	1.31 / 1.48 12.98%	1.90 / 1.85 -2.63%	0.00 / 0.00 0.00%
bc1x Wait (% cycles)	9.99 / 16.55 65.67%	1.43 / 1.74 21.68%	31.92 / 34.21 7.17%	17.76 / 7.29 58.95%
Write Cache Eviction (% cycles)	0.0 / 0.0 0.00%	0.01 / 0.01 0.00%	0.00 / 0.00 0.00%	0.00 / 0.00 0.00%

Table 3.36 Functional Unit Latencies (50M / 1G Instrs. and % Difference)

Functional Unit	ear	fpppp	hydro2d	spice2g6
Load Unit	22.20 / 28.40 27.9%	18.30 / 21.26 16.2%	16.64 / 17.56 5.5%	13.38 / 15.74 17.6%
Store Unit	22.89 / 23.15 1.1%	13.93 / 16.05 15.2%	16.50 / 17.31 4.9%	8.89 / 12.74 43.3%
Add Unit	28.87 / 29.75 3.0%	20.13 / 23.35 16.0%	19.90 / 20.76 4.3%	10.67 / 19.15 79.5%
Multiply Unit	28.55 / 29.70 4.0%	18.01 / 21.10 17.2%	19.21 / 20.01 4.2%	14.64 / 19.77 35.0%
Divide Unit	24.26 / 26.87 10.8%	40.59 / 40.20 1.0%	41.04 / 41.90 2.1%	23.51 / 27.95 18.9%
Conversion Unit	25.12 / 27.68 10.2%	8.86 / 8.99 1.5%	6.64 / 6.76 1.8%	9.41 / 13.37 42.1%
Comparisons	8.18 / 9.17 12.1%	12.71 / 11.02 13.3%	11.03 / 12.04 9.2%	14.58 / 17.34 18.9%
Miscellaneous	27.92 / 29.06 4.1%	8.98 / 7.86 12.5%	8.93 / 9.94 11.3%	5.99 / 15.20 153.7%

Table 3.37 Dynamic Instruction Use (50M / 1G Instruction Run-Lengths)

Instruction	ear	fpppp	hycro2d	spice2g6
FP_LOAD	0.10/0.15 46.08	0.41/0.41 -0.73	0.26/0.26 0.00	0.08/0.02 -78.31
FP_MT	0.01/0.00 -92.31	0.00/0.00 -50.00	0.00/0.00 0.00	0.00/0.00 -100.00
FP_STORE	0.07/0.08 20.29	0.16/0.15 -5.56	0.08/0.08 0.00	0.00/0.00 -50.00
FP_MF	0.00/0.00 0.00	0.00/0.00 0.00	0.00/0.00 0.00	0.00/0.00 0.00
FP_ADD	0.04/0.06 48.65	0.10/0.10 -0.96	0.02/0.02 0.00	0.00/0.00 -100.00
FP_MULT	0.05/0.07 46.94	0.12/0.12 -2.40	0.06/0.06 0.00	0.00/0.00 0.10
FP_DIV	0.01/0.00 -92.31	0.00/0.00 -50.00	0.00/0.00 0.00	0.00/0.00 -100.00
FP_CONV	0.17/0.21 24.70	0.00/0.00 0.10	0.00/0.00 0.00	0.00/0.00 -100.00
FP_COMPARE	0.00/0.00 0.00	0.00/0.00 0.00	0.00/0.00 0.00	0.00/0.00 0.00
FP_ABS	0.00/0.01 55.56	0.00/0.00 0.00	0.00/0.00 0.00	0.00/0.00 0.40
FP_NEG	0.00/0.00 0.00	0.00/0.00 0.00	0.00/0.00 0.00	0.00/0.00 0.00
FP_MOV	0.00/0.00 0.00	0.00/0.00 0.00	0.04/0.04 0.00	0.00/0.00 0.00
FP_BC1X	0.01/0.02 42.86	0.00/0.00 100.00	0.04/0.04 0.00	0.02/0.00 -71.43
INT_LOAD	0.15/0.15 -0.65	0.42/0.42 0.48	0.26/0.26 0.39	0.22/0.20 -8.60
INT_STORE	0.03/0.00 -92.31	0.00/0.00 133.33	0.00/0.00 -12.50	0.06/0.07 21.43
INT_ALU	0.24/0.20 -13.98	0.12/0.11 -6.03	0.26/0.26 -0.39	0.25/0.38 53.23
INT_BRANCH	0.00/0.00 0.00	0.00/0.00 0.00	0.00/0.00 0.00	0.00/0.00 0.00
INT_NOP	0.09/0.07 -30.11	0.04/0.05 23.81	0.09/0.09 0.00	0.20/0.14 -27.18
Avg Difference (for instr.s with a frequency > 5%)	18.74%	2.50%	0.20%	28.19%

cases where simulating only an initial section of a program may lead to inaccuracies. The Aurora III simulator does not make use of sampling, but probably should do so in order to increase both confidence and simulation throughput.

3.11 Evaluation of Final FPU Designs

In this section, the wide range of architectural trade-offs presented in this dissertation are compared via five Aurora III FPU implementations, including a 250MHz baseline FPU, a design which improves CPI through more complex architectural features, a simpler design which achieves a higher clock frequency, and two designs which project the performance gains obtained from the process technology improvements.

3.11.1 Turning Off Architectural Features to Increase Clock Frequency

There are many choices to be made in implementing a processor design, and completely different approaches can often achieve similar performance. This is quite evident in the two prevalent styles for building microprocessors, one which emphasizes the extraction of instruction-level parallelism through greater complexity and the other which focuses on simplifying a design in order to more easily benefit from technology improvements. The Sun UltraSparc processor is typical of the former, while the Dec Alpha embodies the latter design philosophy. Some designs adopt characteristics of both approaches, such as the MIPS R10000. Similar trade-offs were considered for the Aurora III design; it is interesting to consider whether a simpler design might have achieved the same level of performance. Consider the following summary of critical paths (all having a path-length of 20 gates) for the current FPU design:

1. write enable for destination register field of ROB => destination register field of ROB => scoreboard translation table, containing most recent ROB reference
2. add unit exception signals => selection of exceptional constant => write exception field of ROB entry
3. instruction queue head and tail pointers => logic that derives the queue full signal

4. register field for instruction queue entry => scoreboard read port for ROB tag => read port for valid bit of ROB entry

5. size of data in ROB entry => data dependency logic => issue determination => reserve result bus

6. size of data in ROB entry => data dependency logic => issue determination => logic to set scoreboard busy bit and ROB tag

7. size of data in ROB entry => data dependency logic => issue determination => write translation table which supports precise memory exceptions

8. size of data in ROB entry (single or double precision) => data dependency logic => issue determination => signal to stall issue while writing the status register

9. ROB valid => data dependency logic => issue determination => reserve ROB entry (update tail pointer)

10. ROB valid => data dependency logic => issue determination => selection of rounding mode

11. ROB valid => data dependency logic => issue determination => selection of ROB entry to be reserved upon entry

12. ROB valid => data dependency logic => issue determination => reserve store queue entry

13. ROB valid => data dependency logic => issue determination => advance head pointer of load queue

14. ROB valid => data dependency logic => issue determination => signal which initiates an arithmetic comparison

15. ROB valid => data dependency logic => issue determination => signal to initiate multiply

16. integer memory exception => translation table to find floating-point ROB entry corresponding to memory reference => signal to clear all ROB valid bits

17. head entry of ROB generates an exception => logic to clear state throughout FPU and discard instructions that follow the exceptional one

18. counter to time completion of a divide operation => divide unit is free => issue determination => initiate divide

19. external instruction transfer type => predecode logic => create result class, which indicates whether the instruction produces a result

20. external instruction transfer type => logic for writing an instruction queue entry

These paths are representative, but not comprehensive, of the overall set of critical paths in the FPU design and many of these are driven by additional inputs which originate from other sections of the chip. Consequently, there are more parallel paths that derive these signals and are of critical gate depth. For example, since operands may be found either in the register file or reorder buffer, the valid bits for both constructs are used to evaluate whether data dependencies prevent instruction issue; the logic depth is similar for reading the valid bits from both the scoreboard and reorder buffer. Similarly, there are a number of constraints besides data dependencies that are used to determine whether issue is possible and these are not explicitly represented in the above list. Instead, only one path to each output signal is shown. Also, this list does not include any paths internal to functional units, of which there are many.

This list suggests that a number of paths are limited by the logic depth of the scoreboard and reorder buffer. Several designs were evaluated for each of these constructs and the versions that were chosen are close to optimal for the constraints of GaAs DCFL. For instance, a read port on the scoreboard which produces a result in 7 levels of logic is implemented using a stage of multiplexors followed by a tristate gate. Leakage currents in GaAs constrain the amount of fanin that can be tolerated by a tristate gate, whereas in another technology, such as CMOS, several levels could be removed by constructing the read port completely from tristates. In addition, the gate-depth for control logic tends to be greater for GaAs since the DCFL family efficiently supports only NOR-NOR logic. For a complex design which supports most features of the MIPS ISA, the goal of 20 gates per clock phase is fairly aggressive and even the application of significant additional human resources could only reasonably reduce this by approximately 2 gate levels.

So in what ways can the design be made simpler while still attaining the same level of performance? An in-order issue and completion policy would remove the need for a reorder buffer. As mentioned earlier, the reorder buffer provides several benefits; those that are relevant to this discussion are: 1) retaining the in-order sequence of instruction for an out-of-order completion policy, and 2) providing support for precise memory exceptions. The former is not necessary for in-order completion and the latter can be handled by trans-

Table 3.38 Percentage of Memory References that Miss in the IPU Mini-TLB

Benchmark	% Memory References
alvinn	0.191
doduc	0.403
ear	0.195
fpppp	0.081
hydro2d	4.813
mdljdp2	0.664
mdljsp2	0.546
nasa7	1.034
ora	0.002
spice2g6	8.425
su2cor	5.569
swm256	3.175
tomcatv	6.827
wave5	0.271
Avg Arithmetic	2.3

ferring floating-point instructions which follow a memory reference only when it is determined that a page fault cannot occur. Most memory references hit in the small first level TLB that resides in the IPU (see Table 3.38), and will not stall the transfer of floating-point instructions more than a single cycle. As a result, this requirement should have only a modest effect on performance, especially in light of the high integer content of many floating-point programs. For a technology with higher integration levels, MMU functionality would be contained on the same chip as the IPU and all memory access exceptions could be resolved in a single cycle. It is also interesting to note that removing the current reorder buffer would reduce chip area by approximately 20%. The reorder buffer is not as area-efficient as the register file, which is based on a 6-transistor memory cell and sense-amplifier read ports; instead, multiplexors and tristates are used in the reorder buffer for write and read access. The reorder buffer contains 8 entries, each with 89 bits for a total of 712 bits of state,

versus 2048 bits for the register file; still the reorder buffer is 85% larger than the register file. A future version of the FPU should base most or all of the reorder buffer design on the denser register file style.

A scoreboard may still be the most efficient way to resolve data dependencies, but issue logic would be simpler due in part to the fact that dual issue would not be possible; a simple in-order completion policy could not simultaneously issue two instructions to functional units that have different latencies. The omission of a reorder buffer would also simplify the issue logic needed to detect data dependencies, since operands could only originate from the register file. In fact, there are very few constraints that could prevent issue, and all are fairly easy to resolve:

1. an instruction is being executed in a different functional unit than that needed by the instruction to be issued,
2. the current instruction depends on the result of an outstanding instruction,
3. the current instruction uses a non-pipelined functional unit which is busy executing a preceding instruction,
4. for a load instruction, the head entry of the load queue does not contain valid data,
5. for a store instruction, there are no free entries in the store queue.

Together, these changes would allow a new logic depth target of 15 gates per clock phase, which corresponds to an overall reduction in path length of 25%. The register file access time would also need to improve, from the current 1.8ns to something closer to 1.4ns. This may well be attainable since all read/write decoding is currently performed in the same clock phase as the access and it could be moved to the phase that precedes the access, removing 400ps to 500ps from these critical paths. This would need to be explored further.

Critical paths within functional units constitute another issue. It is often difficult to arbitrarily add pipeline latches to a design to increase the clock frequency. So even if path depth for other parts of the design is reduced by a quarter, it may be difficult to do so within

the functional units. For example, consider the 53-bit mantissa adder which contributes to at least one critical phase for each of the functional units. This adder is already pipelined so that the first 3 to 4 gates of the carry and sum logic are generated and then latched. Adding latches elsewhere in the adder can significantly increase area because the logic fans out before converging back to produce a single 53-bit result. Other constructs, such as leading one prediction and sticky-bit logic, can grow substantially in size due to deeper pipelining and more complicated routing. As the area for these functional units increases, all paths are impacted since global routing capacitance increase. As discussed earlier, not pipelining the functional units has only a modest effect on overall performance. Doing this and adding an extra cycle of latency to each unit would allow a higher clock frequency. The analysis which follows will examine these approaches, assuming that deeper pipelining will degrade cycle time by 10%.

The last 2 columns of Table 3.39 compare the two simpler designs which achieve a higher clock frequency to the baseline FPU implementation (column 1). The design which supports pipelined functional units has 19.8% lower performance and the non-pipelined design has 12.2% lower performance. However, the simpler designs would have been easier to implement, resulting in a shorter design cycle for the FPU. The simpler designs could benefit more quickly from process technology improvements since there are fewer critical paths to be reevaluated.

3.11.2 SPECfp92 Comparisons

This section summarizes the current state of microprocessor performance, via SPEC ratings (refer to Table 3.39, Table 3.40, and Table 3.41). In addition, 5 versions of the Aurora III FPU are listed, including the 2 discussed in Section 3.11.1. In addition to the 250MHz baseline design, the other 2 versions are projections of the baseline in light of reasonable technology improvements. The process technology that supports the FPU design has not changed in over 2 years but a newer version should be available soon and it is interesting to speculate on the impact that this will have on the floating-point performance of the Aurora III design. These improvements include both finer interconnect pitches and fast-

Table 3.39 Aurora III SPECfp92 Comparison

Benchmark (50M Instructions)	Aurora III IOOO base 250MHz	Aurora III IOOO base 300MHz	Aurora III IOOO base 350MHz	Aurora III IOIO pipelined 280MHz	Aurora III IOIO non-pipelined 310MHz
alvinn	220.5	264.7	308.8	209.1	231.5
doduc	199.6	239.6	279.5	167.6	184.0
ear	371.7	446.0	520.3	279.6	296.9
fpppp	294.0	352.8	411.6	191.1	210.6
hydro2d	253.1	303.8	354.4	232.9	257.4
mdljdp2	288.4	346.0	403.7	266.1	289.0
mdljsp2	142.6	171.1	199.6	130.9	141.6
nasa7	453.6	544.4	635.1	270.2	299.3
ora	191.6	229.9	268.3	157.4	172.0
spice2g6	138.6	166.3	194.0	141.2	156.4
su2cor	426.5	511.8	597.1	336.0	366.8
swm256	272.4	326.9	381.3	162.5	176.3
tomcatv	369.3	443.2	517.1	264.2	290.7
wave5	171.8	206.1	240.5	153.8	169.2
SPECfp92	253.2	303.8	354.4	203.1	222.3
SPECint92	na	na	na	na	na
% Change from 250MHz baseline		20.0	40.0	-19.8	-12.2

er intrinsic gate switching speeds, which together should decrease the average loaded gate delay by 10% to 40%. It is assumed that register file access time will also decrease, although the amount may not directly track gate switching speeds.

Further, there are a number of features of the FPU that are not accounted for in these simulation-based results, some of which should enhance overall performance:

1) Better code reordering that reflects the unique characteristics of the Aurora III design; the compiler used for these experiments was tailored to the MIPS R2000/ 3000 scalar architecture. This point has been discussed briefly in the context of float-

Table 3.40 SPEC Ratings for Current Microprocessors

Benchmark (50M Instructions)	SGI R8000 (TFP) 75MHz	SGI R10000 200MHz *	DEC 21064 200MHz	DEC 21164 300MHz *	IBM RS/ 6000 580 62.5MHz	IBM Power 2 71.5MHz	Sun SuperSp2 90MHz	Sun UltraSp 167 *
alvinn	793.6	na	436.9	na	206.2	na	na	na
doduc	157.2	na	131.0	na	88.6	na	na	na
ear	596.6	na	587.6	na	174.2	na	na	na
fpppp	279.8	na	193.3	na	172.6	na	na	na
hydro2d	484.6	na	216.8	na	126.7	na	na	na
mdljdp2	290.5	na	153.1	na	124.2	na	na	na
mdljsp2	116.1	na	75.1	na	57.3	na	na	na
nasa7	608.3	na	280.5	na	203.9	na	na	na
ora	236.2	na	156.2	na	103.1	na	na	na
spice2g6	85.2	na	100.2	na	73.7	na	na	na
su2cor	515.4	na	291.9	na	208.1	na	na	na
swm256	361.8	na	226.8	na	95.8	na	na	na
tomcatv	672.6	na	304.6	na	210.3	na	na	na
wave5	180.6	na	115.3	na	69.2	na	na	na
SPECfp92	310.6	600*	200.1	500	124.8	274	147	305*
SPECint92	108.7	300*	132.7	330	59.2	134	135	275*

Designs marked with a "*" have been announced but are not yet commercially available.

ing-point compare sequences; efforts elsewhere have shown performance gains of 20% to 30% [Johnson91] for code that has been compiled for a specific processor implementation.

2) The benefit of having twice as many floating-point registers as the R3000/R4000 microprocessors. This should somewhat alleviate the long 3-cycle primary cache latency by providing more local storage for intermediate results.

3) Branch prediction is not modeled, but as discussed in Section 3.10.1, it should have only a small negative impact on floating-point applications, since basic block size is

Table 3.41 SPEC Ratings for Current Microprocessors, continued

Benchmark	Intel Pentium 815 100MHz	Intel P6 133MHz *	IBM Power 2 71.5MHz	PowerPC 620 130MHz	PA-RISC HP755 99MHz	PA-RISC 8000 200MHz *	AMD K5 100MHz
alvinn	170.5	na	na	na	176.8	na	na
doduc	79.1	na	na	na	142.0	na	na
ear	210.7	na	na	na	258.4	na	na
fpppp	117.5	na	na	na	237.1	na	na
hydro2d	83.0	na	na	na	166.1	na	na
mdljdp2	95.2	na	na	na	192.1	na	na
mdljsp2	44.9	na	na	na	92.3	na	na
nasa7	60.7	na	na	na	123.3	na	na
ora	93.7	na	na	na	276.9	na	na
spice2g6	64.9	na	na	na	91.9	na	na
su2cor	56.9	na	na	na	177.2	na	na
swm256	45.1	na	na	na	79.3	na	na
tomcatv	77.7	na	na	na	138.0	na	na
wave5	55.8	na	na	na	112.1	na	na
SPECfp92	80.6	>200*	274	300	150.6	>550*	105*
SPECint92	100.0	>200*	134	225	80.0	>360*	130*

Designs marked with a "*" have been announced but are not yet commercially available.

quite large.

4) The benefit of supporting double-word loads and stores was estimated in Section 3.7.1; it remains to be verified.

5) Operating system calls are not modeled by a PIXIE-based approach to architectural simulation. For the SPEC benchmarks, the OS is entered on average only 1.5% to 2% of the time; this is less than what is realistic for current multi-media applications.

6) Hardware support for square root can result in a 7% to 9% improvement in performance, but has not been implemented in the current design.

3.11.3 Final design

Appendix A contains a layout plot of final FPU design, which can be summarized as follows:

- 250 MHz, 300 SPECfp92 rating
- 500K transistors
- 16x16 mm2
- 40 instructions (MIPS R4000), including double-word loads/stores and integer multiply
- Iterative 5 cycle Wallace tree multiplier (4-2 adders)
- Pipelined 3 cycle add unit
- Pipelined 2 cycle conversion unit
- Iterative 19 cycle SRT-8 divide unit (not included with this version)
- IEEE-754 compliant (4 rounding modes and exceptions)
- Precise and higher performance real-time exception modes
- Issue policy: in-order issue, out-of-order completion, 2 instructions per cycle
- Data prefetching with unity stride
- Instruction queue: 6 entries, predecoded
- Load queue: 2 entries
- Reorder buffer: 8 entries
- Store queue: 2 entries
- Result busses: 2
- 5 students - 2 years

Verification via random testing

- Functional units verified against actual workstation (>5M operations per unit)
- FPU verified using self-checking random instruction sequences (>200M instructions)

Design-for-test includes:

- 5 scan chains (4 data, 1 control)
- Full read/write access to register file
- Individual instructions are testable via scan paths
- Speed testing requires only the 2 clock signals to operate at high frequency

CHAPTER 4

Implementation of a High Performance Floating-point Unit

The culmination of this work has been the design of an IEEE-754 compliant double precision floating-point unit as part of the Aurora III processor. The chip was designed in a 1.0μm (0.6μm effective gate length) GaAs direct coupled FET logic process. The discussion will focus first on trade-offs for the various functional units that are appropriate for the circuit and integration constraints of GaAs. These constraints include low fanin, greater logic depth due to the use of only NOR-NOR logic, the absence of dynamic logic structures, a limited use of pass-gate logic, and lower layout density that results from a ratioed logic family. Much of the motivation for the functional unit designs originated with work done elsewhere, but has been extended in order to accommodate differences that result from using a high-performance GaAs technology. Also, a number of corrections to the original references were discovered during the verification of these units. Further, the conversion unit that is described is an original design that can execute any of the 6 conversion operations called for in the IEEE-754 specification with a latency of 2 cycles. The second part of this chapter will focus on a set of general issues that arise while designing a floating-point unit, including floating-point loads and stores, the use of predecoding to reduce critical path depth, and reasonable design-for-test features.

4.1 Add Unit

The IEEE-754 compliant double precision floating-point addition unit supports the four rounding modes specified by the IEEE standard: round to nearest, round to ∞, round to $-\infty$, and round to zero. The add unit design is fully pipelined, with a latency of three 4ns

clock cycles, and consists of 50,000 transistors.

4.1.1 Adder Implementation

As discussed in the section on circuit issues, GaAs DCFL presents several challenges compared to CMOS design, including low noise margins, the use of NOR-only logic, and low fanin. In light of these technology issues, a number of different adder implementations were evaluated for the mantissa addition, including carry skip, 4-bit carry look-ahead, and Ling-modified carry-select designs.

The carry-skip adder was not to be appropriate for several reasons [Turrini89], [Majerski67], [Lehman61]. The small fanin of GaAs increases the number of gate levels needed for the skip logic. This is especially true for implementing multiple levels of skip logic, which is necessary to achieve the lowest critical path lengths. Intrinsic gate speeds in GaAs are fast (70 ps for an unloaded inverter), but interconnect and fanout increase the average delay to about 100 ps per level; an adder such as the carry-skip design with greater than 15 levels of logic is unacceptable on a chip with a target of 20 gates per clock phase. The NOR-only limitation is also problematic, since there are instances where the lack of a single-level AND gate leads to additional levels of logic. The carry-skip adder requires that nodes along the carry chain be reset to a logic low value prior to performing an addition. In other technologies, this can be handled efficiently by pre-discharging these nodes. In GaAs this is not feasible because of the high leakage current and the diode gate of the MESFET transistor. Consequently, a mux-based approach is necessary, leading to an increase in the levels of logic. Finally, delay through a carry-skip adder is proportional to the square root of the number of bits, versus a log relationship for a look-ahead adder. As the number of bits increases, there is a cross-over point, beyond which a look-ahead approach is faster; the 53-bit mantissa adder used throughout the FPU is more efficiently implemented using a look-ahead approach. The FPU needs several sizes of adders and it was desirable to choose a single adder design that could be extended to support a variety of bit widths.

The Ling-modified adder is shown in Figure 4.1. The 53 bit wide version in this figure is divided into six blocks, each with a width of 9 bits. Each block is further divided into

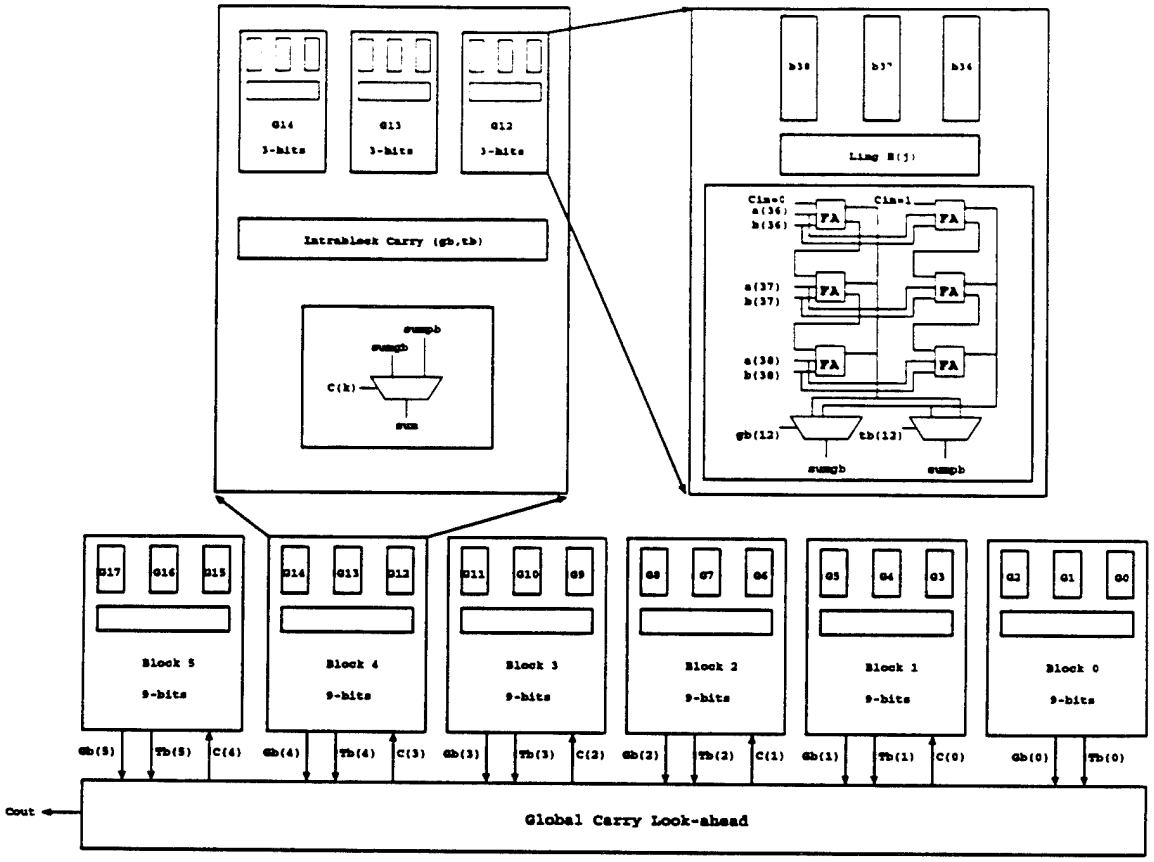


Figure 4.1 53-bit Ling Adder

3-bit groups, with each block having 3 groups. The adder uses a carry-select algorithm within each group and each block. A ripple sum is created for 3-bit groups for both a carry-in of 1 and a carry-in of 0. These pairs of three bit sums are then fed into a multiplexor to form the 9-bit block sums. The block carry generate and propagate signals are used to form the 9-bit block sums. The six block sums are then fed into a second level of multiplexors to create the final sum. The global generate and propagate signals are then used to select the appropriate 9-bit block for the final sum. The group-generate equation is:

$$G_5 = g_{17} + g_{16}p_{17} + g_{15}p_{16}p_{17}$$

The Ling technique [Ling81] uses $g_i = p_i g_i$ to simplify the generate equation:

$$G_5 = p_{17}(g_{17} + g_{16} + g_{15}p_{16})$$

$$G_5 = p_{17}G_5^*$$

$$G_5^* = g_{17} + g_{16} + g_{15}p_{16}$$

Expanding the G_5^* term gives:

$$G_5^* = a_{17}b_{17} + a_{16}b_{16} + a_{15}b_{15}a_{16} + a_{15}b_{15}b_{16}$$

The primary advantage of this adder comes from expressing the local group carry generation in terms of G^*, a signal generated from 4 terms of 10 inputs. Traditional carry look-ahead adders produce group generate signals in 7 terms of 24 inputs. Fanin limitations dictate an optimum group size of 3 bits.

The analysis of critical paths for the Ling-modified adder and a group-4 carry look-ahead adder were based on [Stritter90]. A "direct search" algorithm was used to minimize functions of numerous variables. In the case of the CLA-4 adder, there are 14 independent variables, one for each of the enhancement transistor widths that lie along the top critical path. The algorithm focuses on one variable at a time and evaluates f(x,xi), f(x,xi - delta * ei), and f(x, xi + delta * ei), where x represents all variables not currently being evaluated, xi is the current variable being tested, delta is the step size, and ei is a unit-vector for the variable xi. One of these three expressions will result in the smallest value for the function and this is an indication of a direction that warrants further exploration. Constraints are enforced by adding penalty values to the primary variable when constraints are violated. In the CLA-4 optimization, we for example, calculated the enhancement width for each level to derive the smallest delay for a given power dissipation. Thus, power was a constraint; whenever the calculated widths obtained generated too much power for the path, the overall delay was penalized. This behavior forces the algorithm to move back to transistor widths which do not violate the power budget.

Delay calculation is performed using two-dimensional macro-models [Kayssi93a]. The macro-model derivation results in expressions for delay and output slew rate for each type of gate as a function of fanin, fanout, interconnect capacitance and input slew rate. The form of the expressions are:

Rising output:

$$delta = T_{in} \times (c_0 + c_1 \times y)$$

$$T_{out} = T_{in} \times (c_3 + c_4 \times y)$$

Falling output:

$$delta = T_{in} \times (c_0 + c_1 \times y + c_2 \times \sqrt{y})$$

$$T_{out} = T_{in} \times (c_3 + c_4 \times y + c_5 \times \sqrt{y})$$

Where:

$$y = C_{total} / (WE \times T_{in})$$

$$C_{total} = C_{int} + k_1 \times LW + k_2 \times WE$$

WE = enhancement width of driving gate

WL = sum of enhancement width loads

C_{int} = interconnect capacitance

c0 through c5 are fitting coefficients

There are different coefficients for each type of gate being modeled (i.e., fanin). Interconnect is represented as a lumped capacitance; this approach is quite accurate for the shorter lengths of interconnect seen within a combinational block. The coefficients are obtained by running a fitting algorithm on the results of numerous HSPICE simulations. An empirical relationship for power as a function of enhancement gate width was also obtained using HSPICE.

The first experiments with both the CLA-4 and Ling adders raised several issues. First, the algorithm has a tendency to choose widths that result in large fanouts. Consequently, it was necessary to include a fanout constraint. Whenever the fanout for a certain set of widths exceeds a chosen threshold, the overall delay is penalized. The program then begins decreasing the width of the transistor in question. Second, there was a question about the repeatability of results when the initial starting point is changed. By adding an option to randomly generate the initial widths from a reasonable range, the overall optimized delay

Table 4.1 Comparison of Path-Delay for Optimization Program and HSPICE

Adder Type	Optimization Program (ns)	HSPICE (ns)	Error
CLA-4	1.515	1.659	9.5%
Ling	1.237	1.341	8.4%

was found to be consistent within 3% between runs even though the exact partitioning of delay varied somewhat. Finally, there was a question about the accuracy of delays calculated by this approach. A comparison was made between the prediction of the optimization program and that obtained using HSPICE. For the worst case path through each adder, the results are summarized in Table 4.1. In each case the specified maximum power was 400 mW. The error of less than 10% is quite good, but might be improved. In particular, the coefficients were generated only up to a fanout of four. For larger fanouts, extrapolation was utilized and the non-linear nature of the fitting made the error more significant for these cases.

To meet a 4ns clock cycle, it is necessary to pipeline the adder. This is accomplished by generating and latching the propagate, generate, and local group carry signals during the first clock phase; other work external to the adder would also be performed during this phase. To reduce levels of logic, part of the function that precedes the latch was merged into the first stage of the latch. Figure 4.2 shows the merged NOR-latch which generates the first stage of generate logic for the adder. When the clock changes state, a transition can occur for only one of the two nodes A and B, which feed the cross-coupled pair. Other approaches for merging logic with a latch are possible, but care must be taken to avoid those

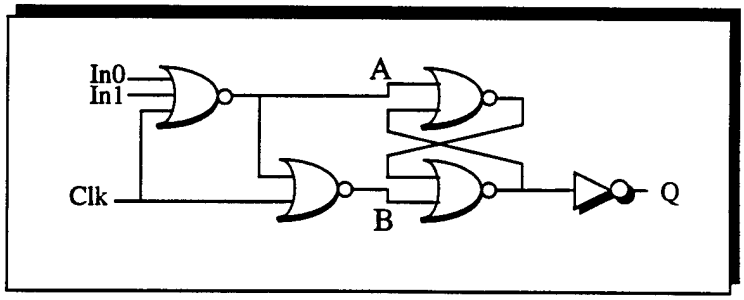


Figure 4.2 Merged Nor-Latch-Buffer Cell

that allow simultaneous transitions on both inputs of the latch pair when the clock changes. This sort of clock hazard is dependent on the layout parasitics of the cell, which, if not carefully controlled, can reduce the yield of this frequently used circuit.

Since rounding is merged into the mantissa addition stage, it is necessary to generate both A+B and A+B+1, requiring two copies of the latter part of the carry generation tree. The sum logic and much of the initial carry generation logic is only implemented once. A second carry chain adds approximately 30% to the overall transistor count of an adder. Round to $\pm\infty$ requires either (A+B and A+B+1) or (A+B+1 and A+B+2,); the latter can be realized using an additional row of half-adders prior to the mantissa (discussed below). The various functional units that in the FPU require 5 different versions of the Ling-modified adder, as summarized in Table 4.2.

A method that detects the potential for a long carry propagation time and that will cause a single cycle stall in order to complete the addition was investigated [Wolrich84]. An upper bound on the probability, P, of generating a stall has been derived as:

$$P = \frac{\frac{-m+w}{n} \times 2^{(-n+w)}}{2^w}$$

where:

w = width of addition

Table 4.2 Ling Adder Implementations Used in FPU

Type of Adder	Delay (ns)	Area (um x um)	Transistor Count	Approximate Power (W)
11 bit	1.01	666x909	2,264	0.328
11 bit, with 2 carry chains	1.02	914x922	2,998	0.418
32 bit	1.3	673x2133	5,913	0.856
53 bit, 2 carry chains, leading one logic	1.6	1445x5210	16,600	2.750
53 bit, 2 carry chains	1.5	1035x3793	13,627	1.998

m = total bits not included in any detection gate (low order bits)

n = width of detection gates

This idea was not pursued further for several reasons. Long mantissa adders are needed in the add, multiply, and divide units and adding the ability to stall would complicate the design for each of these units. In addition, a more complicated and potentially slower acquisition approach for the result busses would be needed, versus the simple reserve-at-issue approach that was implemented. Finally, pipelining the 53-bit mantissa adder removed this component from being a bottleneck along critical paths.

4.1.2 Add Unit Implementation

The algorithm implemented for the add unit takes advantage of several well-known characteristics of floating-point addition (refer to Table 4.3 and Figure 4.3) [Quach90], [Quach91a], [Quach91b], [Quach91c]. For example, the alignment and normalization steps needed for addition are mutually exclusive. If the two operands are both positive, or their signs differ and an alignment shift of more than one is needed, the resulting normalization shift will be less than or equal to one. Conversely, if the signs are different and an alignment shift of less than or equal to one is needed, the normalization shift may be greater than one. An alignment shifter is needed for the former case, while a simple mux can be used to han-

Table 4.3 Floating-Point Addition Algorithm

Pipeline Stage	Operation
S0P1	Exponent Compare Mantissa swap Exponent swap
S0P2	Alignment right shift Sticky bit determination Guard and round bit generation
S1P1	Gen/Prop/Carry for Ling adder Rounding logic
S1P2	Mantissa add (Ling adder) Leading one prediction
S2P1	Complement result Normalization left shift Exponent adjustment Generate exception signals

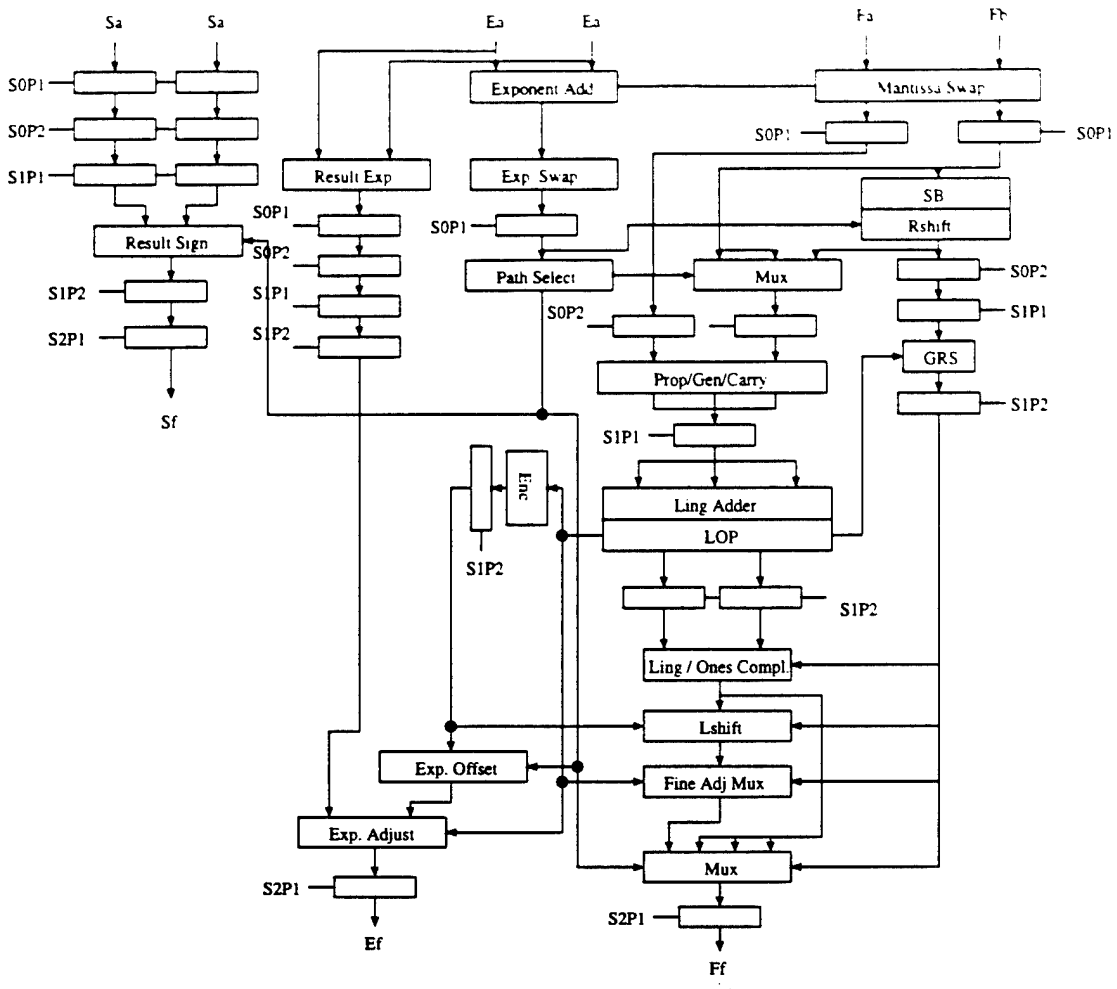


Figure 4.3 Add Unit

dle alignment for the latter case. Normalization also requires a shifter and a mux in order to handle both paths. By implementing two mantissa adders, one for normalization and one for alignment, one can reduce the latency of floating-point addition to only two 4ns clock cycles, as shown in Table 4.4. However, the simulations discussed earlier for the overall floating-point unit architecture show that reducing the add latency by one cycle has a minimal effect on overall performance (approximately 1.5% improvement). Integration levels and yields in GaAs suggest that the 16,000 transistors which comprise a 53-bit adder would be better applied elsewhere.

As indicated in Table 4.3, three operations occur in the first pipeline stage of the add unit. First, the exponents are subtracted to determine which operand is larger and the amount of an alignment shift. The adder used here is 11 bits wide and supports 2 carry

Table 4.4 Two Cycle Add Unit

Pipeline Stage	Normalization Path	Alignment Path
1	Small alignment shift Mantissa add	Large alignment shift
2	Normalization	Mantissa add Small normalization shift

chains in order to generate both A+B and A+B+1. The exact functions produced by this adder are:

$$ExpAlign_0 = Exp_A + \overline{Exp_B}$$

$$ExpAlign_1 = Exp_A + \overline{Exp_B} + 1$$

The amount of alignment shift needed can then be generated by:

if $ExpAlign_{carryout}$ shAmt = $ExpAlign_1$
else shAmt = $\overline{ExpAlign_0}$

The mantissas are then swapped based on the carry-out of the ExpAlign adder in order to ensure that the smaller operand is aligned to the larger one.

In the next stage, S0P2, the alignment shift occurs via a 55-bit shifter which also generates the guard and round bits. This shifter is implemented as a cascade of two 8-input super-buffer muxes, where the mux selects are simply the output of the preceding exponent adder. Note that more area intensive squeeze muxes were used in order to minimize the gate depth through the shifter, since several of the bits of the shifter are used along critical paths for both this phase and the normalization phase. The two muxes of this shifter can shift by (0, 1, 2, 3, 4, 5, 6, or 7) and (0, 8, 16, 24, 32, 40, 48, or 56) respectively; the shifter is comprised of 4,989 transistors and returns a result in 500ps. The sticky-bit is also generated during this phase through the use of a thermometer function. This logic works by creating a vector in which the number of low order bits that are set equal the alignment shift amount. This vector is then AND'ed with the mantissa being aligned; the result is a new vector which represents those bits that are shifted off the low order end during alignment. An OR tree is used on this new vector to create a single result which indicates whether any of the bits below the least-significant-bit of the aligned mantissa are set (excluding the guard and

round bits, which are dealt with separately). This approach to generating the sticky bit is more efficient solution than the direct approach of using a 106-bit shifter. Because generation of the sticky-bit is not on a critical path, the logic to generate it was synthesized from a behavioral description of the pertinent equations. Some of the initial logic needed for rounding determination, including the generation of the guard and round bits, is also produced in this pipeline stage. In verifying this design, I discovered several errors with the approach described in [Quach91a]; the final equations for the guard and round bits are listed in Appendix B.

In the next stage, S1P1, the local group-generate signals for the carry tree and the initial XOR results for the sum are produced and latched. As described earlier, this acts to split the operation of the mantissa across two clock phases. The inputs to this stage of the adder are the aligned operands or one of several transformed variations of the operands. The most common of these is the complement of one of the operands when the operation being performed is subtraction. A somewhat more complex case involves the two round-to-infinity modes (round-to-minus-infinity, RM, and round-to-plus-infinity, RP), which require the mantissa adder to produce three sum results: A+B, A+B+1, and A+B+2. To understand why this is necessary, consider adding the following two mantissas under the RP mode (this example is abbreviated to 5 bits for clarity):

```
11000 0  operand A, whose exponent is twice as big as that for operand B
01000 1  operand B, which has been shifted right one bit for alignment
-------
100000 1  right-most bits are the lsb (L) and the guard bit (G); the sticky bit (S) is zero
```

The intermediate result needs to be shifted right by one for normalization and then rounded by one to satisfy the RP mode. However, rounding has been merged into the mantissa addition phase to reduce latency and to allow the use of only one 53-bit adder. This means that to round properly, a one must be added to the bit position just above the lsb, instead of at the lsb position (anticipating the subsequent normalization). In other words, the adder must be able to produce A+B+2. Interestingly, this is not needed for the round-to-nearest (RN) mode since in this case the rounding one is added to the lsb only when L=1,

G=1, S=0; consequently, adding a one at the lsb position will result in a propagation which does not depend on whether a right shift occurs. This is the tie case for RN (i.e., in decimal, 5.5 would be considered a tie, which for a similar binary example would be rounded up only if the lsb is set); for the situation where L=1, G=1, S=1 and a 1 bit normalization shift is required, a rounding one added at the lsb will suffice due to propagation. The question arises of how to generate the three different sums for RP without adding yet another carry tree. To generate A+B+2, a row of half-adders is inserted into the pipeline prior to the generate/ sum stage of the adder, as shown in Figure 4.4. The A+B+1 result can be created simply by setting the lsb of the adder output to one, since this case will only be needed when the lsb is zero.

An optimization that has been alluded to, is to round during mantissa addition instead of during a separate step. The rounding logic determines which of the adder outputs to select and is derived during both this stage and the subsequent S1P2 stage. Much of this logic has been implemented by hand since it is on a critical paths for this phase. Several signals are derived from 64-entry truth tables which were optimized using 6-input Karnough maps. The mux-reduction technique described in Section 5.9 was used extensively to factor out late arriving signals, such as the carry-out of the mantissa adder. A number of bugs were identified during verification and led to modifications for the equations derived in [Quach91a]; the final equations are summarized in Appendix B.

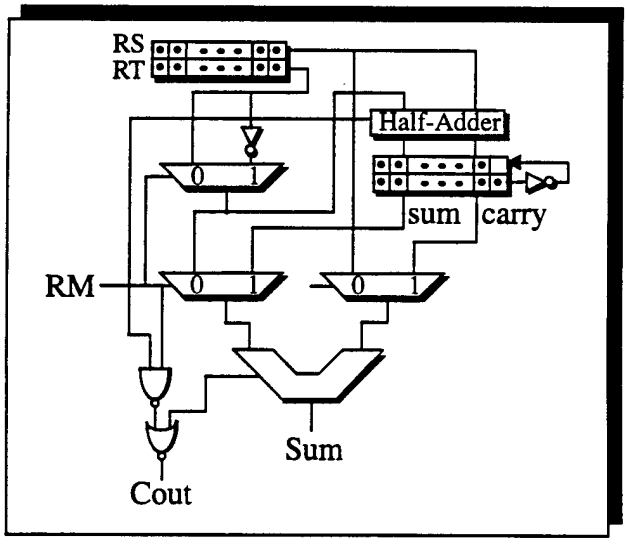


Figure 4.4 Generating A+B+2 for RM/RP Rounding Modes

During the next stage, S1P2, the size of a normalization shift is determined in parallel with the mantissa addition. While leading one prediction (LOP) requires additional logic, it does not need to await the completion of the addition. As mentioned, during the alignment phase the two exponents are compared, and the mantissas are swapped if required to ensure that the result will be positive. However, if the exponents are equal, the result may still be negative and the LOP logic must be able to detect either a leading one (positive) or a leading zero (negative) in the result. An alternative would be to perform a magnitude comparison and swap the mantissas during the alignment stage. The first approach results in slightly more complicated LOP logic, while the second requires an additional alignment shifter since the magnitude comparison will be done concurrently with the actual alignment; to minimize area considerations, the former approach was chosen.

The LOP logic was initially based on [Quach91b] and can be broken down into three stages. First a vector Ubar is generated from the two inputs to the adder. The first occurrence of a zero in Ubar (from the most significant bit) indicates the position of the leading one/zero. The equations that define this vector represent all cases that have the ability to generate a leading one/zero from the two inputs. In this discussion, the following conventions are assumed:

Ai and Bi are the ith bits of the input operands A and B

Ti = XOR(Ai,Bi)

Zi = NOR(Ai,Bi)

Gi = AND(Ai,Bi)

The expression that was originally used is equation (4) from [Quach91b]:

$$U_i = T_{i-1} + \bar{T}_{i-1}\left(\left(\overline{T_{i-2} \oplus A_{i-1}}\right) Z_i + \left(T_{i-2} \oplus A_{i-1}\right) G_i\right)$$

However, random verification found a number of cases not covered by this equation and a new relationships for U_i was derived as the OR of the following terms:

$$T_{i+1} Z_i \overline{G_{i-1}} \qquad G_{i+1} G_i T_{i-1} \qquad Z_{i+1} G_i \overline{G_{i-1}}$$

$T_{i+1}G_i\overline{G_{i-1}}T_{i-1}$ $\quad$ $G_{i+1}T_iZ_{i-1}$ $\quad$ $Z_{i+1}\overline{G}_iT_{i-1}$

$T_{i+1}G_iG_{i-1}$ $\quad$ $G_{i+1}Z_iG_{i-1}$ $\quad$ $Z_{i+1}T_iG_{i-1}$

$\overline{T_{i+1}}Z_iG_{i-1}$ $\quad$ $G_{i+1}G_iZ_{i-1}$ $\quad$ $Z_{i+1}T_iZ_{i-1}$

$G_{i+1}T_iT_{i-1}$ $\quad$ $Z_{i+1}Z_iG_{i-1}$

$G_{i+1}Z_iT_{i-1}$ $\quad$ $Z_{i+1}G_iT_{i-1}$

These 16 equations contain a certain degree of redundancy and can be reduced to an OR of the following seven:

$Z_i(T_{i+1} \oplus G_{i-1})$ $\quad$ $G_{i+1}(Z_i\overline{Z_{i-1}} + \overline{Z}_i\overline{G_{i-1}})$ $\quad$ $Z_{i+1}T_i\overline{T_{i-1}}$

$T_{i+1}G_i\overline{Z_{i-1}}$ $\quad$ $Z_{i+1}T_{i-1}$

$Z_{i+1}(G_i \oplus G_{i-1})$

These reductions make use of certain characteristics of T, Z, and G:

$$\overline{G}_i = T_i + Z_i$$

$$\overline{Z}_i = T_i + G_i$$

$$\overline{T}_i = Z_i + G_i$$

In addition, the most-significant bit of Ubar is generated by the following relationship:

$$Ubar_{52} = T_{52}Z_{51}Z_{50} + G_{52}G_{51}Z_{50} + Z_{52}G_{51}Z_{50} + Z_{52}T_{51}G_{50} + G_{52}T_{51}G_{50}$$

The cell used to generate the Ubar vector consists of 20 gates and 9 inputs. This large number of inputs results in the cell being wire limited and the effective height of the cell (including spillover routing from the overcell region) exceeds all other datapath cells by 15 microns. As a result, this cell sets the datapath pitch for one-half of the FPU, adding an additional 6% to the overall area of the chip. Additional design effort devoted to this cell should be able to resolve this problem.

Ubar is converted, through the use of a parallel CLA-like tree, into a second vector, "SH", which contains a single one at the position of the leading one/zero. This conversion is reflected in the following equation:

$$SH_i = AND(Ubar_{[N:i]}, \overline{Ubar_{i-1}})$$

The parallel look-ahead reduction is shown in Figure 4.5 and the variables are defined as:

Glop0 = AND(Ubar53,Ubar52,Ubar51)

.

.

Glop17 = AND(Ubar2,Ubar1,Ubar0)

GBlop0 = AND(Glop0,Glop1)
GBlop1 = AND(Glop0,Glop1,Glop2)

.

.

Glop10 = AND(Glop15,Glop16)
Glop11 = AND(Glop15,Glop16,Glop17)

GGlop0 = AND(GBlop1,GBlop3)
GGlop1 = AND(GBlop1,GBlop3,GBlop5)
GGlop2 = AND(GBlop1,GBlop3,GBlop5,GBlop7)
GGlop3 = AND(GBlop1,GBlop3,GBlop5,GBlop7,GBlop9)

This one-of-53 SH vector is reduced to the 6-bit encoded shift amount (Elop) that controls the normalization shifter. The defining equations are:

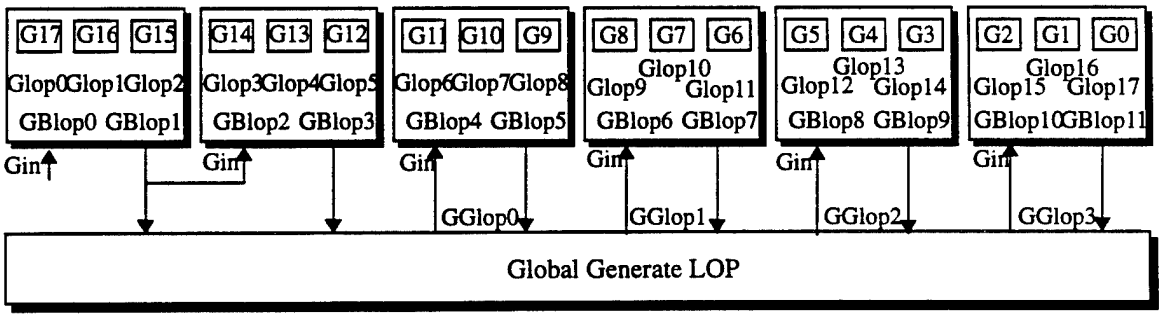


Figure 4.5 Generation of SH Vector for LOP

```
Elop(0) =
    OR(SH{1,3,5,7,9,11,13,15,17,19,21,24,25,27,29,31,33,35,37,39,41,43,45,47,49,
    51})
Elop(1) =
    OR(SH{2,3,6,7,10,11,14,15,18,19,22,23,26,27,30,31,34,35,38,39,42,43,46,47,5
    0,51})
Elop(2) =
    OR(SH{4,5,6,7,12,13,14,15,20,21,22,23,28,29,30,31,36,37,38,39,44,45,46,47,5
    2})
Elop(3) =
    OR(SH{8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31,40,41,42,43,44,45,46,47
    })
Elop(4) =
    OR(SH{16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,48,49,50,51,52})
Elop(5) =
    OR(SH{32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52})
```

The Elop vector is used to predict the leading-one position for normalization, but it will be accurate only to within one bit position. In other words, it may be necessary to perform a 1-bit fine adjustment shift using a multiplexor.

Several actions occur during the last stage, S2P1. First, if no alignment was necessary, the output of the adder may be negative, since no magnitude comparison was performed on the mantissas. Consequently, this intermediate mantissa result may need to be complemented. At this point, any large normalization shift is performed using the value of Elop generated from the leading-one prediction logic. This normalization shifter is similar in organization to the alignment shifter, using a cascade of two 8-input multiplexors. The final mantissa selection depends on several things, including whether the alignment or normalization path was selected, the decision of the rounding logic, and whether an additional 1-bit normalization is needed. The last condition, derived by looking at the most-significant bit of the normalization shifter, sets the effective logic depth along this path at 20 gates. The term "effective" is used here because the actual gate depth is less than 20, but one of the gates must drive a signal across a 53-bit column of the fine adjustment multiplexor. In all

other parts of the FPU, multiplexor select signals are derived during the phase previous to when they are actually used. A buffered gate can drive the capacitance and fanout encountered for a 53-bit column in 300 to 400 ps, or approximately 3 to 4 gate delays. The critical path just described could be reduced by adding logic in parallel to the mantissa adder in order to determine whether a fine adjustment shift will be needed.

Also during this phase, the exponent (Ef) is adjusted to reflect the outcome of any normalization. The various possibilities are summarized in Table 4.5. The actions indicated in this table are implemented using an 11-bit exponent adder which has two carry chains and can produce A+B and A+B+1. A number of the signals which generate the offset for this adder and which determine the result that is selected arrive late in the phase. To mitigate the effect of late-arriving signals, this logic was hand-generated like the rounding logic.

Finally, information about exceptions and the form of the result is generated during this phase. Several bits encode whether the result is infinite, not-a-number (NaN), or zero. These bits, used during the reorder buffer write phase, select either the computed result from the add unit or one of the three constants. The same circuitry is used for the other functional units. Generating these bits in the reorder buffer write phase works well because circuitry used in the final phase of most functional units has the critical logic depth, and it cannot accommodate the additional selection logic.

Table 4.5 Generation of Final Exponent

Normalization Class	Ef
Many Left Shift (MLS)	Efi - Elop
Many Left Shift and Fine Adjust	Efi - Elop - 1
One Left Shift (OLS)	Ef1 - 1
No Shift (NXS)	Ef1
One Right Shift (ORS)	Ef1 + 1

4.1.3 Comparison Instructions

The MIPS ISA specifies a broad range of comparisons for floating-point data via three possible conditions: unordered (either operand is a NaN), equal (the result mantissa is zero), and less than (the result is negative). The various predicates are encoded into the lower 3 bits of the instruction and are implemented by subtracting one of the specified operands from the other. Instead of producing a data result, they generate a single bit which indicates the outcome of the comparison; this bit is written into the floating-point status register via the reorder buffer. In addition, this bit of the status register is exported from the FPU, along with a condition-valid signal, to be used by branch-on-FPU instructions.

4.1.4 Functional Verification

The add unit design was fed random floating-point numbers in each of the three main operating regimes: alignment equal to zero, equal to one, and greater than one. The same random operand pairs were also fed to a verification program running on the host workstation and the results of both were compared. In this way the results of several hundred thousand computations were verified, with a performance of 1.5 calculations per second. At this point, a compiled-code version of the add unit was used to improve the simulation time to 30 calculations per second and more than 10 million new calculations were performed.

4.2 Conversion Unit

The MIPS ISA supports conversions between any of the 3 number formats (integer/word, IEEE-754 single, and IEEE-754 double precision floating-point), which result in 6 possible operations:

cvt.d.s	single to double
cvt.d.w	word to double
cvt.s.d	double to single

cvt.s.w	word to single
cvt.w.s	single to word
cvt.w.d	double to word

A commonly encountered sequence of instructions involves changing the rounding mode and then converting an operand to the integer format. Like the MIPS R4000, the Aurora III FPU adds the following instructions, which encode the rounding mode directly into the instruction to eliminate the need to change the control register before and after the conversion:

ceil.w.s ceil.w.d	same as cvt.w but for the round-plus-infinity (RP) mode
floor.w.s floor.w.d	same as cvt.w but for the round-to-minus-infinity (RM) mode
round.w.s round.w.d	same as cvt.w but for the round-to-nearest (RN) mode
trunc.w.s trunc.w.d	same as cvt.w but for the round-to-zero (RZ) mode

Initially, we intended to implement conversions in the add unit to save logic. However, many of the paths in the optimized add unit are already at the limit of 20 gates per phase, especially in the rounding logic. After further investigating the operations and logic needed to support conversions, it became clear that merging the two sets of instructions into the add unit would adversely impact the unit's speed. The final design for the conversion unit includes a modest 30,000 transistors.

For the discussion which follows, refer to Figure 4.6 which shows the block diagram of the conversion unit. The organization of the unit evolved from first identifying the operations and exceptions associated with each of the conversion types. The following summarizes these actions, describing first the steps involved in the conversion and then the possible exceptions that may result:

cvt.d.s

1. Shift mantissa left from 24 bits to 53 bits. Since floating-point operands are normal-

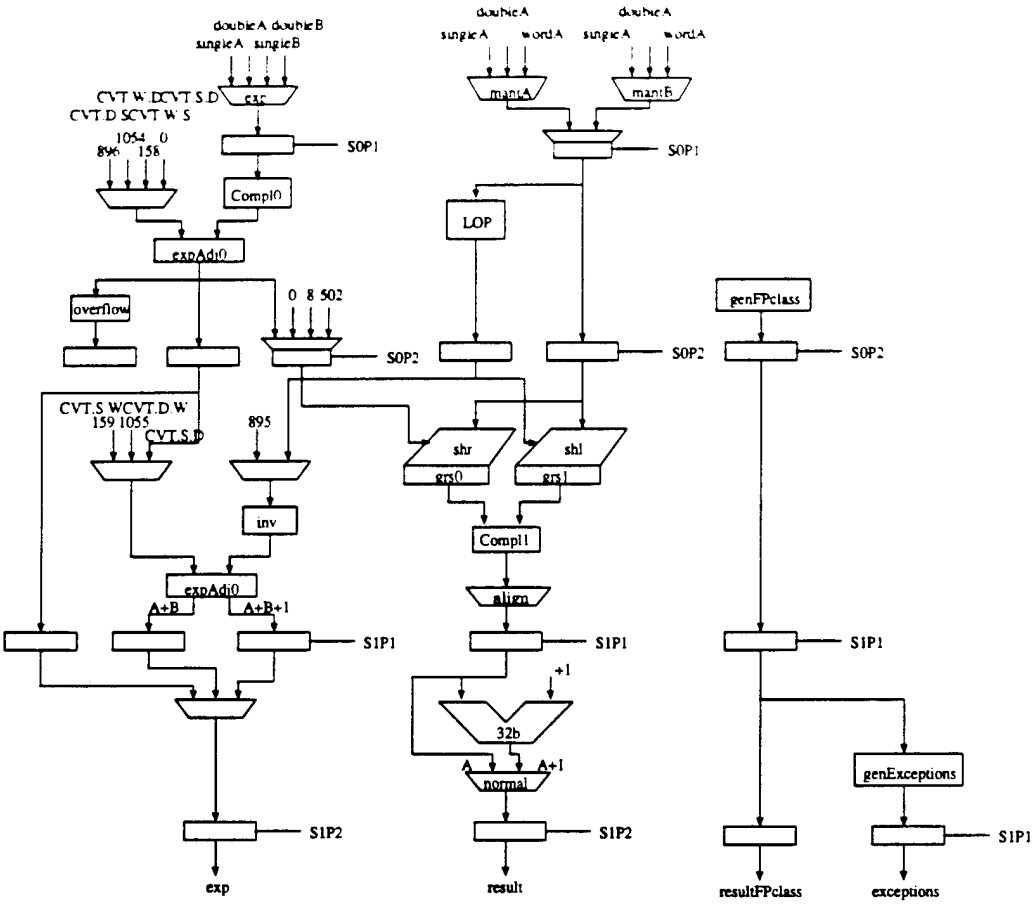


Figure 4.6 Conversion Unit

ized to 1.xx, this shift, performed using a multiplexor, will always be by a constant number of bits.

2. Re-bias the exponent: $e_d = e_s + 896_{10}$, where e_d and e_s refer to the 8 bit and 11 bit exponents for single and double precision. The subscript "10" for "896" means that this number is of base 10; some numbers for the discussion that follows may be represented in hexadecimal, or base 16. The constant "896" results from subtracting the bias for a single precision number ("127") from the bias for a double precision number ("1023").

Invalid:	Operand is a signalling NaN and flag is enabled. A quiet NaN operand will produce a quiet NaN result if this flag is not enabled.
Inexact:	Cannot occur, since no rounding is needed.
Underflow:	Cannot occur, since the range and precision of the double precision format is larger than that for single precision numbers.
Unimplemented:	1. Operand is NaN and Invalid flag is disabled.

2. Source operand is a denormal.
3. Instruction attempts conversion from double to double, which is not allowed.

Overflow: Cannot occur, for same reason as Underflow.

cvt.d.w

1. Normalize mantissa by shifting right until the msb is a one. This will be done with leading-one prediction logic and a left shifter.
2. If the two's-complement operand is negative, complement the normalized result in order to obtain a sign-magnitude representation.
3. Generate A+1 ('A' being the mantissa result from step 2) in order to complete the two's-complement inversion of step 2.

4a. Select between A and A+1 depending on the sign of the input operand.

b. Generate the result exponent: $32_{10} - (shAmt + 1) + 1023_{10} = -shAmt + 1054_{10}$

Invalid: Cannot occur, since there is no representation for a signalling NaN in the word format (although a quiet NaN is possible).

Inexact: Cannot occur, since no rounding is needed.

Underflow: Cannot occur.

Unimplemented: 1. Operand is NaN and Invalid flag is disabled.
2. Instruction attempts conversion from word to word, which is not allowed.

Overflow: Cannot occur.

cvt.s.d

1. Derive the guard, round, and sticky bits.
2. Add a rounding bit if necessary.

3a. Adjust exponent: $e_s = -e_d + 896_{10}$

b. Normalize mantissa by one and add one to exponent, if necessary.

Invalid: Operand is a signalling NaN and flag is enabled.

Inexact:	1. If guard, round or sticky bits are set. In other words, the result is not the same as an infinitely precise result. 2. If an overflow has occurred.
Underflow:	If enabled and tininess (result falls into the denormalized range) or if disabled, tininess, and loss of accuracy (guard, round, or sticky bits are set).
Unimplemented:	1. Source operand is a signalling NaN and Valid exception is not enabled. 2. Source is denormalized. 3. Instruction attempts conversion from single to single, which is not allowed. 4. Underflow has occurred.
Overflow:	If overflow occurs. Care must be taken to examine the result after a possible 1 bit normalization is performed.

cvt.s.w

1. Normalize mantissa, using leading-one logic and left shifter.
2a. If the operand is negative, complement the normalized result.
 b. Generate guard, round, and sticky bits.
3. Produce A, A+1 using 32 bit mantissa adder.
4a. Depending on sign of operand, rounding mode, and guard/round/sticky bits, select between the results of step 3.
 b. Generate the exponent: $-(shAmt+1)+32_{10}+127_{10} = -shAmt+158_{10}$.
5. Normalize mantissa by one and adjust exponent, if necessary.

Invalid:	Cannot occur, since there is no signalling NaN for the word format.
Inexact:	If guard, round, or sticky bits are set.
Underflow:	Cannot occur, since the range and precision of the single format is large enough to represent any 32 bit two's complement integer.
Unimplemented:	1. If a source operand is a signalling NaN and the Valid exception is not enabled. 2. Instruction attempts to convert from single to single, which is not allowed.
Overflow:	Cannot occur.

cvt.w.d

1a. Shift mantissa right by $-((-1023_{10}+e_d)+1)+32_{10} = -e_d+1054_{10}$. If the amount of the shift is less than zero, then an overflow has occurred. If the amount of the shift is greater than 31, then the shift amount should be set to 31; this means that the operand is less than 1.0.

b. Derive the guard, round, and sticky bits.

2. If the operand is negative, complement the result from step 1.

3. Generate A, A+1 using a 32 bit adder. The incremented version will be used for adding either a complementing or a rounding one. It is necessary to add one or the other but not both, as will be discussed below. Select between the two results based on the sign of the input operand, the rounding mode, and the guard/round/sticky bits.

Invalid:	1. Source operand is infinity. 2. Source operand is a signalling NaN and this exception is enabled. 3. An overflow has occurred.
Inexact:	If any of the guard, round, or sticky bits are set.
Underflow:	Cannot occur.
Unimplemented:	1. Source operand is a signalling NaN and the Valid exception is not enabled. 2. The operand is denormalized. 3. Instruction attempts to convert from word to word, which is not allowed. 4. Underflow has occurred.
Overflow:	If an overflow has occurred.

cvt.w.s

1a. Shift mantissa right by $-((-127_{10}+e_s)+1)+32_{10} = -e_d+158_{10}$. If the amount of the shift is less than zero, then an overflow has occurred. If the amount of the shift is greater than 31, then the shift amount should be set to 31; this means that the operand is less than 1.0.

b. Derive the guard, round, and sticky bits.

2. If the operand is negative, complement the result from step 1.
3. Generate A, A+1 using a 32-bit adder. The incremented version will be used for adding either a complementation or a rounding one. It is necessary to add one or the other but not both, as will be discussed below. Select between the two results based on the sign of the input operand, the rounding mode, and the guard/round/sticky bits.

Invalid:	1. Source operand is infinity. 2. Source operand is a signalling NaN and this exception is enabled. 3. An overflow has occurred.
Inexact:	If any of the guard, round, or sticky bits are set.
Underflow:	Cannot occur.
Unimplemented:	1. Source operand is a signalling NaN and the Valid exception is not enabled. 2. The operand is denormalized. 3. Instruction attempts to convert from word to word, which is not allowed. 4. Underflow has occurred.
Overflow:	If an overflow has occurred.

Several observations can be made about the organization of Figure 4.6. First, all conversion types have a latency of only 2 cycles. Architectural simulations have shown that conversions are used infrequently and have little impact on overall performance, so a short latency may not be of tremendous importance. However, a uniform latency does simplify various parts of the issue logic, including reserving a result bus. Second, rounding will never require adding both complementing and rounding ones to the mantissa. The reason is related to the nature of rounding. Rounding is necessary whenever there are too many bits of precision in an intermediate result to fit into the width of a format. In other words, for a normalized number, there are bits "hanging off the end." In order for a complementing one to propagate from these extra bits up into the final section of the mantissa, these additional bits must be all zero prior to the complementation. This means the guard, round, and sticky bits are all zero and it is not necessary to round the result. Thus, only A and A+1 need to be

generated but never A+2, simplifying the design of the mantissa adder. This observation is also used to simplify the rounding logic in the add unit.

The preceding summary of conversion types shows that several instructions may require a 1-bit normalization shift after the mantissa adder. This normalization step, which is implemented with a multiplexor, is on a critical path. Waiting for the adder result before generating the select signals for the multiplexor causes unacceptable delay. The two reasons for adding a one to the mantissa are, as just discussed, complementation (C_c) and rounding (C_r). A C_c is used for both instructions that convert from an integer operand to a single or double precision result. These conversions will never require a 1-bit normalization shift, since there is no way for a one to propagate from the most significant bit of the adder. In order to do so, the operand after complementation would need to be all ones; this corresponds to an input operand of zero, which would not need to be complemented in the first place. On the other hand, a C_r can propagate out of the msb. Consider the following three simplified cases:

A	B	C
100	110	111
+1	+1	+1
-----	----	----
101	111	1000

The results of cases A and B do not need to be normalized, but that of C does. Because case C arises only when all bits of the input to the adder are set, this condition is easily detected in parallel to the actual addition, reducing the logic along the critical normalization path. The cvt.d.w instruction will never need this normalization since it can add a complementation one (C_c), but not a rounding one (C_r). The cvt.s.d and cvt.s.w instructions both utilize a rounding C_c and benefit from this approach.

Leading-one detection logic is somewhat simpler for the conversion unit than for the add unit, since there is only one input operand. The top bit of the Ubar vector, $Ubar_{31}$,

is zero. For lower-order bits, the logic looks at 3 bits simultaneously (although the third bit will be shown to be unnecessary) and creates a vector "Ubar", where the first bit set from the left indicates the position of the leading one. The various possibilities are shown in Table 4.6.

The defining equation is: $\overline{(\overline{(A_{i+1}+A_i)}+\overline{(\overline{A_{i+1}}+\overline{A_i})})} = Ubar$, which is simply an XOR of A_{i+1} and A_i. The Ubar vector, which may contain several logic-ones, is translated in a vector, "SH", which has only one bit set at the position of the leading-one. This in turn is translated into a 5 bit binary encoding which is used to control the normalization shifter. As in the add unit, the SH vector is generated using a parallel look-ahead approach in order to reduce the gate-depth of this logic and meet the target of 20 gates per phase. The equation which defines SH is: $SH_i = AND(Ubar_{[N,i]}, \overline{Ubar_{i-1}})$. The final version of this design was verified using random operands with more than 10 million vectors.

4.3 Multiply Unit

Conventional approaches to multiplication involve a partial product array (3-2 or 4-2 carry-save adders) followed by a carry-propagate mantissa adder. 4-2 carry-save adders create a more area-efficient layout than 3-2 adders by providing both a regular placement

Table 4.6 Leading One Prediction for the Conversion Unit

3 bit window $A_{i+1}A_iA_{i-1}$	$Ubar_i$ comment
000	0
001	0 since this will give a shift of 1 too few
010	1
011	1
100	1 since a negative number will also need a normalization shift
101	1
110	0 since this will give a shift of 1 too few
111	0

of cells and simpler routing. Booth recoding of the input operands can be used to reduce the number of levels in the array by one, at the expense of adding recoding muxes. A comparison of Booth and non-Booth approaches shows that the transistor count is lower for the Booth recoded version, but the area is greater due to the fact that the routing is more complicated (refer to Table 4.7). Increases in capacitance along critical paths of a Booth recoded multiplier can offset the reduction in logic depth. This conclusions was reached by creating a structural description for each design. However, instead of fully implementing the designs, we used bounding boxes to represent the datapath leaf cells. This approach enabled a fast turn-around for placement, routing, and timing analysis. Only after evaluating the different designs, did we design the cells for the iterative non-Booth multiplier that was finally implemented. For comparison, the fastest CMOS double precision multiplier to date generates a result in 8.8ns [Makino93].

The other alternative evaluated was the iterative use of a smaller array. This approach reduces the size of the multiplier considerably, although it requires 5 cycles to produce a result. Such a multiplier is blocking, in the sense that additional multiply instructions must wait for the currently executing instruction to complete. A block diagram of this non-pipelined implementation, which was used in the Aurora III FPU, is shown in Figure 4.7. As discussed earlier, the 2-cycle pipelined multiply unit results in a 10% improvement in performance, but it is too costly in terms of area given the integration constraints for GaAs; the faster unit would have accounted for almost a third of the overall chip area. Integer multiplication is also performed in the multiply unit. The final version of this design was verified using random operands with more than 10 million vectors. The design and analysis of

Table 4.7 Multiplier Implementations

Design	Area (mm2)	Transistors	Transistors per square mm	Delay (ns)
(4-2) Non-Booth	32.41	138,565	4,275	7.70 (2 cycles)
(4-2) Booth	39.75	118,432	2,979	7.67 (2 cycles)
(4-2) Iterative, Non-Booth	24.19	94,501	3,907	20.0 (5 cycles)

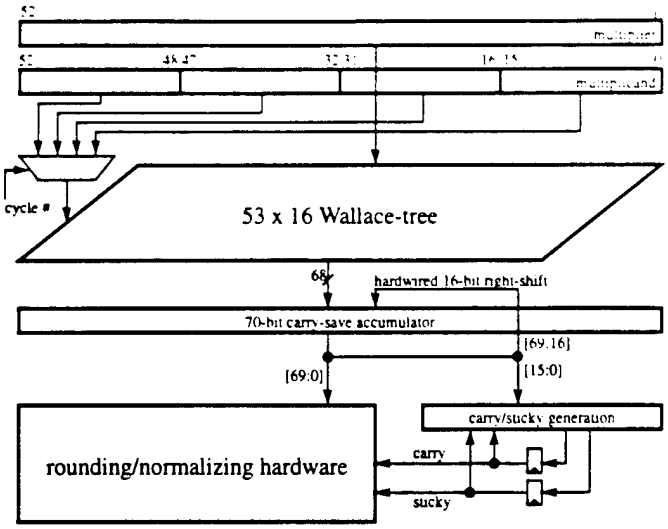


Figure 4.7 5 Cycle Iterative Partial-Array Multiply Unit

the multiplication unit was done by Mike Riepe and is described further in [Riepe93], [Riepe94].

4.4 Divide Unit

As described in Section 3.8.1, non-restoring division algorithms can be combined with the representation of intermediate results in a higher radix redundant form. SRT-2, SRT-4, and SRT-8 approaches generate 1, 2, and 3 bits per cycle, respectively, and latencies vary from 20 to 50+ cycles. Additional techniques can be employed to perform on-the-fly conversion from redundant form to sign-magnitude form and on-the-fly rounding of the result. A square root instruction can be fairly easily mapped to the same hardware used for division, with little increase in area. The design and analysis of the divide unit was done by Dave Putti and is described further in [Putti93]. The divide unit was not included in the current implementation of the FPU due to area limitations; future improvements in process technology should allow the divide unit to be added to the design.

4.5 Precise Exceptions

Being precise, with regard to exceptions, means that the machine state at the time of the exception is the same as for a sequential CPU model; all instructions issue and com-

plete in order. A formal definition might be [Iacobovici88], [Smith88]:

1) All instructions prior to the interrupting one have completed.

2) All subsequent instructions are unexecuted and have not modified state. For more ambitious architectures, this may result in a need to restore the program counter, status registers, and RF operands.

3) If an instruction causes an exception, the program counter points to the address of this instruction for use by the exception handler.

There are two classes of exceptions, the first relating to floating-point computation and the second concerning memory faults; since they are constrained by different issues they will be discussed separately.

4.5.1 Floating-point Computation Exceptions

There are two issues related to precise exceptions: how much should the IPU slip ahead of the FPU and how much can be done in parallel within the FPU. The latter is handled effectively by a reorder buffer. However, with the use of an instruction queue, the former can be quite costly, since the IPU may have slipped far ahead of the FPU. To maintain precise exceptions, it would be necessary to back-up the state of the IPU to the instruction after the exceptional one, at the expense of much storage logic and a possible increase in critical path lengths. The simplest approach for synchronizing the IPU and FPU is to have floating-point operations always stall the integer pipeline. A better variant of this option involves exception prediction, where parts of the operands are compared to certain constants to determine if an exception is possible. As an example, if the biased exponent field of both single precision operands is less than 192, an overflow will not occur. A requirement to be precise will always impact performance and a conservative prediction policy costs more in performance.

There are several implications in not supporting precise floating-point computation exceptions: 1) a greater burden is placed upon the software, and 2) it is more difficult to restart a program after an exception. The latter concern would be a problem for real-time

systems which must always be able to produce a correct result and cannot allow the termination of a program, such as the control system for an airplane. However, the environment of a workstation is somewhat more tolerant and the primary concern here is that a program will abort after running for many hours. Several techniques can be used to reduce the negative effects of imprecise exceptions while still taking advantage of the potential performance gains seen by not being precise. First, the additional range that an extended-precision mode offers might reduce exceptions for sections of code that are required to be reliable. However, support for these less frequently utilized extended formats may both increase and complicate the resources devoted to floating-point functionality. The approach used in the Aurora III FPU implements a separate precise mode of operation, where setting a bit in the IPU and FPU control registers ensures that a precise execution model is followed. After one or two instructions are transferred to the floating-point instruction queue, the FPU asserts the queue-full condition, relaxing it only when each transferred instruction has completed without generating an exception. The IPU will not proceed until the FPU has determined that the instructions are exception-free. In addition to this method, check-pointing of some type might be used to periodically save the state of the processor, so as to allow restarting. If an exception were encountered, the precise mode would be enabled and the program would be restarted from the last checkpoint. A trap handler would then be called when the exceptional instruction is reached, and a result would be returned by software routines.

The great majority of stable, well-tested applications do not experience floating-point exceptions; for example, there are no computation exceptions found in the tens of millions of cycles for the SPECfp92 benchmarks. If a program shows a tendency toward encountering exceptions, the code may need to be rewritten to detect exceptions before they occur. In a sense, implementing precise exceptions is not consistent with the RISC philosophy of emphasizing speed and justifying hardware expenditure based on frequency of use. For instance, page-faults are a common occurrence and require precise handling; floating-point exceptions are quite rare and so one should not spend significant complexity or resources implementing them. It is interesting to note that several commercial machines have

been unintentionally imprecise, including the IBM 360/91, Cray, and CDC 6600/7600, and that a growing number of current machines offer a high performance mode which does not support precise floating-point exceptions, including processors such as the RS/6000 from IBM and the R8000/TFP from MIPS/SGI.

4.5.2 Memory Exceptions

Memory exceptions are a different class from floating-point computation exceptions, in that a computer must be precise with respect to page faults. Whereas floating-point exceptions occur infrequently, memory exceptions are a part of the normal operation of a program and must preserve the in-order sequence of instructions. In part, this means that floating-point loads and stores must reserve an integer reorder buffer entry, as do integer loads and stores. If a page fault occurs for a floating-point load or store, the exception field in the integer reorder buffer entry for the corresponding instruction will be marked. When this instruction reaches the head of the reorder buffer, an exception is signalled and processed. All subsequent instructions are flushed from the reorder buffer. The state of the FPU upon detection of a memory exception must be consistent with that of the IPU; in other words, no instructions after the exceptional one can be allowed to change the state of the FPU (write the register file or control register). In order to implement this, all load and store instructions, regardless of whether they are integer or floating-point, must be sent to the FPU. In addition, the tag corresponding to the integer reorder buffer entry must accompany the instruction. When the integer load/store instruction reaches the head of the instruction queue it will issue within the FPU and will be allocated a floating-point reorder buffer entry. When it is known that a load/store will not cause a page fault, both the integer reorder buffer tag and a valid signal will be sent to the FPU. Using a small lookup memory, the correct floating-point reorder buffer entry will be marked as valid. This approach requires two exception-signalling pins between the IPU and FPU, one for when the IPU recognizes a memory fault and one for when the FPU recognizes a floating-point numerical exception. In addition to the exception signal, the IPU must send to the FPU the tag of the integer reorder buffer entry that caused the exception, since this same instruction may not yet have

reached the head of the floating-point reorder buffer. To put this another way, the FPU may not have completed executing all instructions that occurred prior to the exceptional one. The FPU must finish these outstanding instructions before any additional instructions are transferred. After catching up to the IPU, the FPU must perform the following:

1. all scoreboard valid bits are cleared since all instructions after the exceptional one will be discarded,
2. all reorder buffer valid bits are cleared,
3. all result bus valid bits are be cleared, in order to ensure that a reorder buffer valid bit is not subsequently and erroneously set,
4. all head and tail pointers are reset, including those for the reorder buffer, instruction, load, and store queues.

The FPU ensures that no additional instructions are transferred by simply asserting the queue full condition until the internal state of the FPU has caught up to the IPU. In the meantime, the IPU is free to begin executing integer instructions from the memory fault trap handler. As discussed earlier, all compare and store instructions must pass through the floating-point reorder buffer prior to writing state, in order to ensure a precise model of execution for memory faults. Finally, the IPU (and LSU) must continue to retire any valid store queue entries since these will correspond to instructions that occurred prior to the exceptional one.

The MIPS ISA also requires that reads and writes of the floating-point status register must stall until all issued instructions have completed and have written back to the register file. The former constraint is necessary since there is no additional hardware to restore the contents of the status register if an exception occurs and a subsequent instruction has changed the register. Access of the status register is rare (2.2% of dynamic instructions, on average across the SPECfp92 benchmarks) and does not justify the cost of additional recovery logic. Passing the data to be written to the status register through the reorder buffer in order to maintain a precise execution model is difficult, since several fields in the status register (rounding modes, exception enables) are fed directly to the various functional

units. Waiting for the status register data to reach the head of the reorder buffer before committing the write would mean that subsequent instructions might not be executed with the correct modes. Bypassing these fields directly from each reorder buffer entry would be costly. The constraint of stalling the issue of a status register instruction until all outstanding instructions have written the register file is necessary since a result in the reorder buffer may cause an exception, but will not be detected until it reaches the head of the reorder buffer.

4.6 Implementing Floating-point Loads

The determination of whether load data is valid occurs some number of cycles after a floating-point load instruction has been transferred to the FPU. Data for a load miss will actually be sent twice to the FPU; the first time occurs prior to knowing whether a data cache hit has occurred and the data is sent directly from the cache via the dcOut bus. Later, when the miss data is either received from the BIU/MMU or is retrieved from the write cache it will be sent to the FPU using the dcIn bus. These different events require a means of applying the validation signal to the correct load queue entry. One approach would require the load-store unit to maintain information about which load queue entry an outstanding load instruction has been allocated. This tag would need to be sent along with the validation signal in order to ensure that the correct load queue entry is marked as valid. The load-store unit can store the address of the tail entry of the load queue (a new floating-point load instruction always allocates the tail entry of the load queue), since it is the IPU that initiates pushes and pops to the load queue. In other words, the load-store unit can have a duplicate, independent copy of the Lq tail pointer, and at any given time, the FPU and load-store unit copies should be the same. For a load hit in the data cache, this load queue tag will stay in the load-store unit pipeline until the tag comparison has been made, at which time the tag will be sent to the FPU along with the validation signal. For a load miss, the load queue tag will be written to the miss-status-holding register (mshr) that corresponds to the floating-point load instruction. When the load data is returned from the BIU/MMU, this tag will be read from the mshr and sent with the data and validation signal to the FPU.

A preferred option for guiding load data and validation signals to the correct load queue entry will be presented here. Similar to the approach for precise memory exceptions described in Section 4.5.2, when a floating-point load instruction is transferred to the FPU, the corresponding integer reorder buffer tag is also sent. The entry number in the load queue for the floating-point load being issued is written into another small tag memory (LqTag-Mem). When data from a load miss arrives from the BIU, or when data previously written to the FPU is known to have hit in the data cache, the integer reorder buffer tag for this memory reference is resent to the FPU along with data and validation signal. The tag memory is read to obtain the correct entry number in the load queue. This tag memory must have as many entries as there are integer reorder buffer entries. This approach was selected for the Aurora III design because it requires minimal additions to the existing load-store unit functionality.

In a number of cases, the IPU needs to send an integer reorder buffer tag to the FPU. These are when:

a) writing LqTagMem when a floating-point load instruction is transferred to the FPU,

b) writing the load queue with data from the data cache (occurs 2 cycles after the instruction was transferred),

c) writing a load queue valid bit for a data cache hit (happens in the second cycle after the tag was returned),

d) writing the load queue with data and the valid bit for a write cache hit or a load miss that has been returned via the BIU.

These lead to the following constraints:

1. *a* and *d* cannot happen simultaneously, since load data sent to the FPU via dcIn is prioritized higher than the transfer of floating-point instructions.

2. *(a* or *d)*, *b*, and *c* can occur simultaneously, which calls for a small integer reorder buffer tag pipeline in the FPU, to track the progress of load data through the data cache; this approach requires only one external tag bus between the IPU and FPU.

3. *a* requires write access to LqTagMem.

4. *b*, *c*, *d* all need simultaneous read access to LqTagMem. This will require 3 read ports for LqTagMem.

5. *b* and either *a* (for move-to-FPU instructions) or *d* need simultaneous write access for load queue data. Consequently, the load queue will need to have 2 write ports.

6. *c* and either *a* (for move-to-FPU instructions) or *d* need simultaneous write access for load queue valid bits. Two valid-bit write ports will be needed for the load queue.

4.7 Implementing Floating-point Stores

Store data will be written into the store queue in the same order it occurs in the program; in other words, there is no need to reorder the stores as they are extracted from the store queue by the load-store unit. In addition to the result data, the store queue will also contain a store-type designator and integer reorder buffer tag field (both of which will need to be dedicated pins). There are two distinct classes of store instructions, those that send data to memory (swc1/sdc1) and those that transfer data to the integer register file (mfc1/cfc1). Only the swc1/sdc1 instructions will write data to the write cache. An alternative to sending the store type and destination field from the FPU to the IPU would be to have a small queue in the load-store unit. This queue would be written when the floating-point store instruction is transferred to the FPU. However, this queue would need to have as many entries as the possible number of outstanding floating-point stores (the number in the instruction queue plus the number in the store queue plus one for the store unit pipe stage). The designer is faced with a choice between a few additional pins addition of a fairly large memory structure. To minimize chip area, because of the limited integration levels of GaAs, I chose the former approach for the Aurora III FPU.

The main complexity in implementing floating-point store instructions concerns the fact that the data is not ready when the write cache entry is allocated (or overwritten for a store write cache hit). For an integer store, the procedure is to allocate a new write cache entry if the store misses in the write cache. The address and data are written at this point

while the rest of the line is written when the data is available from the data cache or secondary memory. Since the floating-point data is not ready at the time the write cache entry is allocated, either the data or the rest of the line may appear first. In addition, more than one floating-point store may hit in the same write cache entry (and word within that entry). It is important to not mark the entire line as valid if one or more floating-point stores are still outstanding. The load-store unit also needs to know to which write cache entry the floating-point store data to be written. In the discussion that follows, it is assumed that each write cache entry has separate valid bits for each word of the line, and there are 8 words per entry. An "fpbusy" valid bit is also needed for each word. The sequence for handling floating-point stores is as follows:

1. The store instruction and address are forwarded from the ALU stage of the IPU to the first stage of the load-store unit. If the address hits in the write cache but the fpbusy flag is set for this word, the instruction is set aside (this will be described below in more detail). Otherwise, the floating-point store instruction is transferred to the FPU.

2a. On a write cache miss, a write cache entry is allocated and the address is written to the entry. A lazy writeback policy ensures that there is a free entry; an instruction will not be forwarded to the load-store unit if all write cache entries have words outstanding. The word valid bit is cleared (unless this is an integer store, in which case the word valid would be set) and the fpbusy bit is set.

b. On a write cache hit, the fpbusy bit is set and the word valid bit is cleared. If the fpbusy bit was set, another floating-point store to the same word in this line is outstanding.

3a. For a write cache miss, the line will either be in the data cache or will need to be fetched via the BIU/MMU. In either case, when the line is returned each word valid bit is set, unless the fpbusy bit is set. The data for a word of the entry should not be written if the valid bit is already set (this happens if a previous instruction was an integer store, or if the FPU has returned the outstanding floating-point store data prior to the line arriving).

b. At some point, either before or after 3a, the store queue will be loaded with the required word. A separate dedicated tag bus from the FPU to the IPU will contain the integer reorder buffer tag for the floating-point store at the head of the store queue (this tag was sent initially along with the instruction to the FPU). This tag from the FPU will then be used to retrieve the address from the integer reorder buffer entry, which in turn will be sent to the data cache. At the same time, the data will be sent to the data cache from the FPU via dcIn, and will also be sent to the IPU via dcOut so it can be written into the write cache. If there are no other outstanding stores to the same word of this line (the determination of this will be discussed below), the fpbusy flag is cleared and the word valid flag is set. Only if all word valid flags are set can this write cache entry be written back to the data cache and the secondary memory system. Note that it is possible for the write cache to fill up without being able to write back an entry; this would occur when all entries have outstanding floating-point stores. When the write cache is full, the load-store unit must not accept any more store instructions until an entry becomes free.

A potential problem arises when a given entry in the write cache has more than one outstanding floating-point store to the same word. The processor must be able to determine whether an entry in the store queue is the last one to reference that word in the write cache A solution to the problem might involve assigning an "fpbusy" tag to each outstanding store; this tag is similar in nature to those used in the reorder buffer to write-back data to the register file. In the reorder buffer write-back case, there can be multiple entries in the reorder buffer, all of which write the same register; the scoreboard must not be cleared until the most recent register reference reaches the head of the reorder buffer. This is accomplished by storing the reorder buffer tag for the most recent instruction in the scoreboard entry for the destination register. When an entry reaches the head of the reorder buffer, the scoreboard is cleared only if the number of that entry matches the scoreboard tag for the destination register. A similar approach could be applied to the floating-point store problem, which is in reality a reordering problem. The size of the tag would need to accommodate the maximum number of stores that can be outstanding at a time. When the write cache

entry is allocated, or a word in an entry is overwritten during a write cache hit, this tag would be written to a field in the entry for that word. When the data was returned from the store queue, the fpbusy tag in the entry would be compared to the fpbusy tag sent along with the data from the FPU. If the two were the same, the fpbusy valid bit for that word of the write cache entry would be cleared and the word valid bit would be set.

As an alternative, the fpbusy tag could be replaced with an fpbusy count field for each word in a write cache entry. This count field would be incremented or decremented each time an outstanding floating-point store for this word is issued or retired, respectively. The count field needs to be large enough to accommodate the maximum number of possible outstanding floating-point stores. However, if the count field uses a Johnson encoding, the incrementing/decrementing could be done without an adder, at the expense of needing more bits of storage for each count field. The fpbusy valid bit would be cleared and the word valid bit set only if the fpbusy count is zero for a store retired from the store queue.

Both of these approaches add significant complexity and resources for a condition which will seldom be encountered. A final option would also involve setting an fpbusy bit whenever a floating-point store allocates a write-cache entry or writes to an existing entry. However, if another store to the same word arrives at the load-store unit while the first one is still outstanding, the corresponding instruction and address would be placed into a single entry holding register. It is necessary to set aside the instruction in this way, since upon completion of the first floating-point store, the address of this first instruction must be re-sent to the load-store unit from the integer reorder buffer as the data is sent to the IPU via the dcOut bus. This action needs access to the first pipe stage of the load-store unit in order to rederive the address of the write cache entry. In addition to setting aside the second store instruction, the load-store unit will signal the ALU stage to not forward any additional instructions until the second store has been activated, after the first store has completed and written the write cache entry. This approach is preferable to the first two for a number of reasons. First, there is no longer a need for additional state in the form of an fpbusy tag or count fields for each word of a write-cache entry. Second, the case of having two or more outstanding stores write the same word should occur infrequently, therefor a solution

should not require a significant increase in either complexity or chip area. A load instruction which hits a word in the write cache while a floating-point store is outstanding to this word must also follow this set-aside procedure. Such a load cannot progress further since the data it needs is not yet available. The mechanism of setting aside load and store instructions which reference the same word of a write-cache entry ensures that memory references function reliably in the Aurora III processor.

4.8 Predecoding Floating-point Instructions

Critical paths in the FPU design are well balanced and no section of the design exceeds the goal of 20 gates per clock phase. The issue logic, in particular, is fairly complex, having its operation constrained by: data dependencies via the scoreboard and reorder buffer, invalid load queue entries, busy multiply and divide units, unavailable result busses, an empty instruction queue, a full reorder buffer, a full store queue, and status register accesses. After issue has been resolved, a host of other actions may need to be performed, including: setting the scoreboard valid and tag fields, reserving a result bus entry, initiating a multiply or divide operation, reserving a reorder buffer entry, selecting the source for operands, and advancing to the next instruction queue entry. The combination of decoding and evaluating the two instructions at the head of the queue could easily exceed the target gate path depth. Since several pipe stages are required for floating-point instructions to reach the queue, there is ample opportunity to derive a set of predecoded signals which can simplify the logic needed by the issue clock phase. If the predecode logic were placed in the IPU, additional pins would be required to transfer the instructions. Instead, instructions are predecoded in the phase immediately before they are written into the queue. This time slot occurs naturally in the Aurora III design; if the predecoding were performed earlier, this phase would be wasted. The use of predecoding does require an additional 15 bits per entry in the instruction queue. The predecoded signals can be summarized as follows:

1. Iclass: a 3-bit tag that indicates which functional unit will be utilized by this instruction.

2. Idepclass: a 2-bit tag that refers to how many source operands are used by the instruc-

tion (0, 1, or 2).

3. Iresclass: a 1-bit tag that indicates whether the instruction produces a result that will need to pass through the reorder buffer.
4. FUspec: a 3-bit tag, written to a result bus shift register, identifies which functional unit will write the reorder buffer upon completion of the instruction.
5. FUvector: a 5-bit valid vector, written to a result bus shift register, aligns data properly to be written into the reorder buffer.
6. IclassExcept: a 1-bit signal that identifies certain types of exceptions, such as an invalid opcode or an operand format that is illegal for a given instruction.

The predecode logic is duplicated in order to handle the 2 floating-point instructions that can be transferred to the FPU per cycle. Table 4.8 summarizes characteristics of this logic and shows that removes as many as 12 gate levels from the issue phase.

4.9 Design-For-Test Features

Design-for-test is often added near the end of a design, and designers are usually concerned about how much area and design time will be needed. Full scan is impractical for an integration-poor technology like GaAs DCFL. Every design-for-test feature added must be thoroughly tested, both to ensure it will do what is intended and to ensure that it

Table 4.8 Predecode Logic Statistics

Signal	Synthesized Logic Depth
Iclass	10
Idepclass	8
Iresclass	10
FUspec	10
FUvector	12
IclassExcept	12

does not introduce errors into a stable design. With these constraints in mind, I included the following features in the FPU:

1. Two scan chains for the register file. The register file is sense-amplifier based; the yield associated with this analog design and dense memory structures in GaAs is a concern. The two scan chains provide accessibility to the register file, one for the 5 address ports (1 write, 4 read) and one for the input and output data. In addition, special write enable and clock signals have been added to allow the normal write logic to be bypassed when the FPU is being tested.

2. A scan chain for the 23 main issue-logic signals. The observability gained with this scan chain should one to identify a problem in some component of the FPU that prevents issue from occurring. These signals are not controllable, since they originate from parts of the design that are difficult to access without adding design complexity, such as the valid bits for the reorder buffer and scoreboard.

3. A scan chain for one of the two result busses, allowing the result of any functional unit to be verified prior to writing the reorder buffer and the register file.

4. A scan chain for the top 2 entries in the instruction queue, including all predecode signals.

5. A test signal, testStallFPU, which ensures that the issue of instructions will stall until the instruction and load data queues have been loaded. In conjunction with some of the other test features, this allows any instruction and operands to be loaded, executed, and verified.

6. External access to clock and reset signals on the distribution network. This provides easy verification of basic functionality during initial testing.

Together, these additions increase the chip area less than 1% while providing greatly improved access to internal points in the design.

4.10 Hardware Support for Denormals

A denormalized number occurs when a result exponent is too small to be represented by a given floating-point format. The IEEE 754 specification requires that the returned denormal be the infinitely precise result multiplied by a large constant and then appropriately rounded. These steps can be accomplished without impacting cycle time or overall chip area by using a state machine to perform the necessary mantissa shift and exponent adjustment. However, the additional design and verification complexity was not seen as consistent with the goals of an academic project. In addition, denormals occur rarely and can be handled in software. In fact, the most common occasion for denormals arises during iteration convergence when a result value is less than some tolerance threshold; this is often produced by a subtraction and a subsequent comparison to zero. The fact that a denormal might result is not particularly significant since for all purposes the difference is really comparable to zero. The MIPS R4000 ISA addresses this case by implementing a flush-to-zero mode in which denormalized results are set to zero. The Aurora III FPU follows this approach and will trap to a software handler if the mode is not set.

CHAPTER 5
CAD Support for High Performance Designs

5.1 General Observations on CAD for VLSI

During the design of a chip, there often arises a need to solve a specific problem by creating a custom CAD tool. For example, a utility might be needed to determine the fanout for each gate in a design and alert the designer to any instances which exceed a certain threshold. There are a number of trade-offs to consider when implementing these tools, including what programming environment to use, how often the tool will be used, what size of a design the tool will be used for, and how long it will take to create and verify the tool.

In many cases, the tool will do nothing more than read an ASCII file, such as a netlist, and perform some transformation; for this type of problem a script language such as "awk" or "perl" will suffice. For example, it might be necessary to add an attribute to critical nets in order to ensure that the corresponding interconnect wires have a certain width, and hence no more than a certain value of resistance. Scripting languages, which offer support for manipulating text files, are easy to use, allowing for a fast design cycle.

There is a trade-off between how many times the utility will be run and how fast it needs to operate. A delay-calculator will be run many times throughout the design cycle, whereas a tool to derive the power dissipation of a chip may only be used several times. In the latter case, a scripting language might be acceptable, even if it results in a runtime on the order of several hours. However, for applications that require a fast runtime, a low-level programming environment, such as C or C++, will allow the programmer to better tune the utility for performance.

Care must be taken in how an algorithm is implemented and there are a number of issues to consider. The use of recursion often provides the most efficient implementation, but can make verification somewhat more difficult. It is important to keep in mind the size and complexity of the designs that the utility will address. This will often determine whether to use a linked-list or a hash table. Hash tables tend to require more effort on the part of the programmer but are often invaluable in managing the data structures for a large design. Linked-lists are easier to implement but should be used only in cases where their length will allow efficient traversal. Often a combination of hash tables and linked-lists offers the best solution. Dynamic allocation of memory for data structures is another important CAD issue. Because of the cost associated with calling allocation routines, it is important to exclude the use of these routines from sections of code which are executed frequently. Data structures should be created only once and subsequently accessed with pointers.

The nature of a CAD algorithm itself may limit performance. The levelizer described below has several modes of operation, one of which can lead to runtimes of several days for designs having large amounts of connectivity. The issue here is not that the algorithm has been implemented poorly, but instead, that in order to achieve a reasonable runtime, a feature of the utility may need to be disabled. In many cases, the most important issue concerns how long it takes to develop the tool. The tool will address a specific issue and in so doing will facilitate a stage of the design process; often it is difficult to proceed with a design until the tool has been completed. Careful work in the early implementation stages can minimize the iterations required to improve the performance of the tool.

5.2 Delay Calculation

Delay calculation for the Aurora III methodology uses 2D interpolation tables based on work by [Kayssi93a]. The tables are generated by running HSPICE for each primitive cell with various combinations of interconnect capacitance and fanout. Since the building blocks in GaAs DCFL are fairly limited, primitive cells consist only of an inverter, NOR gates of fanin between 2 and 4, and super-buffer cells which have a fanin between 1 and 8. Information about both the delay and rise/fall time (slew) for a gate are calculated. For the

plain DCFL gates the axes of the lookup table are:

$$\left(\frac{C_{Total}}{Slew \times W_{driver}}\right), \left(\frac{W_{driver}}{W_{driven}} = Fanout\right)$$

C_{total} is the sum of the interconnect capacitance and an empirically derived value for gate capacitance. This latter relation involves the width of the driver, the width of all driven gates, and whether the transition is rising or falling; this rise/fall information is necessary because current flows into the gate of a MESFET transistor, and gate capacitance varies with gate current. For super-buffers, the axes are simply the total capacitance and the input slew rate.

Traversal of a design uses a recursive routine provided by Cascade Design Automation. The algorithm begins at all primary outputs and works backward to each primary input using the following approach:

```
for (i=0; i<num_outputs; i++) calc_delays(inst[i]);
calc_delays (inst) {
  for (j=0; j<num_inputs(inst); j++) {
      if (inst.input[j].slew == -1 && inst.input[j].net != primary_input)
          calc_delays (inst.input[j].driver);
      delay = interpolate(inst.input[j]);
      if (delay > inst.delay) inst.delay = delay;
      }
  }
```

When the data structure for the circuit is created, the various values for each instance are initialized. For example, the worst case delay through a gate will be the input-output pair with the maximum delay; initially the delay for the gate is set to zero. Similarly, the slew for each gate is initialized to minus-one. Referring to Figure 5.1, this simple example would be traversed as follows:

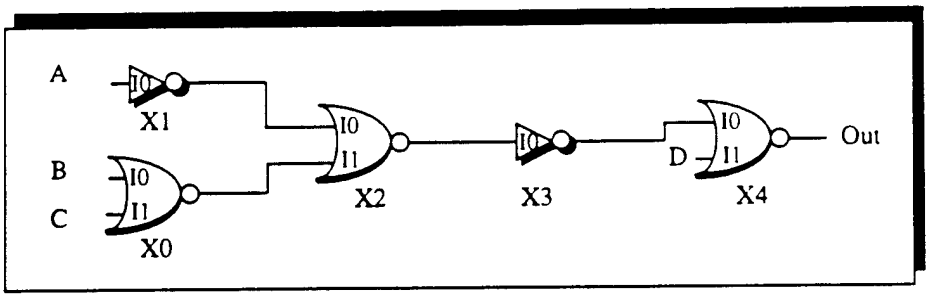


Figure 5.1 Recursive Network Traversal

1. visit X4.I0
2. visit X3.I0
3. visit X2.I0
4. visit X1.I0, calculate delay, slew
5. return to X2.I0, calculate delay, slew
6. visit X0.I0, calculate delay, slew
7. visit X0.I1, calculate delay, slew
8. return to X2.I1, calculate delay, slew
9. return to X3.I0, calculate delay, slew
10. return to X4.I0, calculate delay, slew
11. visit X4.I1, calculate delay, slew

Each instance may be visited more than once, but by utilizing the point at which a delay is actually calculated, each input on a gate will be visited only once. This engine for traversing a design is the core for several of the utilities which follow.

In order to verify the accuracy of the delay calculator, several utilities were created to automatically generate a ready-to-run sensitized HSPICE netlist from a path selected within the Cascade timing analyzer (Tactic). First, a program based on the traversal engine of the delay calculator is run to generate an ASCII database for a design. This database con-

tains information about each instance, such as the interconnect capacitance at the output, the width of the transistors that comprise the gates, and the net names for all inputs and outputs of each instance. A second program reads this database and creates an appropriate data structure for the design. At this point, paths that appear in Tactic can be processed extremely rapidly and an HSPICE netlist for each path is created. A script is then used to run HSPICE and compare the simulated results to those produced by the delay calculator. For the Aurora II design, several hundred paths of various lengths (from a few gates to greater than 30 gates) were evaluated. In all cases the error was found to be less than 10% and in the majority of cases (>70%) the error was less than 7%. The difference came primarily from two sources. First, there is an inherent error in both the mathematical fit involved in generating the interpolation tables and the interpolations. Second, the approach to delay calculation just described does not consider the effect of more than one input being high for a multiple input NOR gate. This situation results in an output node being discharged slightly faster than for the case where only a single input is driven by a logic one. Neither of these error sources is very significant since the overall accuracy is within 10%.

Delays derived by these routines are used in two ways. First, a version of the delay-calculator is directly linked into the static timing analysis environment. Information about large delays, capacitances, and fanout can be dynamically reported to the user. Dracula-modelled capacitances can be incorporated into the calculation routines, as will be discussed below. Second, the delay-calculator can run in a stand-alone mode and the results can either be back-annotated to any digital simulation environment (Verilog, VHDL, Mentor Graphics) or used to drive a buffer sizing/selection utility.

5.3 Capacitance Extraction

The extraction of parasitics is an important part of any methodology for performing timing analysis. Parasitics may need to be extracted on both the local cell level and the higher global level. For the former, source code for a local cell extractor was obtained from Cascade Design Automation and was modified to support the additional interconnect layers available in the Vitesse GaAs process. The results from the local extractor were verified

using DRACULA and the error was less than a few percent. Global interconnect extraction, however, proved to be more difficult. In order to support a fast iteration for timing analysis, the present Cascade tools do not precisely extract global interconnect. Instead, a single empirical capacitance value (in femto-farads per micron) is derived for each interconnect layer and the overall capacitance for a net is found by simply multiplying this layer constant by the net length that is routed in that layer. This approach can, in the worst case, lead to significant errors in capacitance. Consider a case where two wires run from adjacent sources to adjacent destinations. One wire follows an overcell track across a datapath and encounters a large amount of interlayer capacitance, whereas the other wire is routed in a channel and sees only the capacitance to the substrate and ground plane. In most process technologies, interlayer capacitance for long wires, especially for adjacent layers, is more significant than the capacitance down to the substrate or up to the ground plane. The calculated value will be the same for both wires, when in fact one wire may have a capacitance up to 100% larger than the other. The results of delay calculation and subsequent decisions about critical paths, buffering, and wire sizing can all be affected by inaccurate capacitances.

Global extraction is further complicated by how the empirical value for each layer is derived. Some heuristic percentage is applied to the plate and fringing capacitance of each layer, and the sum of these derated values across all layers defines the single empirical value for that layer. However, the nature of these heuristics are included in the router and are not visible to the user. As a result, the layer capacitance values in the process file must be adjusted until the calculated capacitances are in the best agreement with Dracula generated ones. This was done by using two representative test cases, one comprised of standard cells and the other of datapath cells. A script was written which reads the Cascade database and adjusts the various layer capacitance parameters until the best match with Dracula is found. Table 5.1 contrasts the original process file values (obtained from Cascade) with those derived from this approach. Using the improved layer values results in an average error of about 30% and a maximum error of about 90%. These errors are due to the inherent inaccuracy in the approach for global extraction, but are acceptable for first-pass timing analysis.

Table 5.1 Global Capacitance Tuning

Test Case	Avg Difference with Dracula (%)	Largest Difference with Dracula (%)	Largest Capacitance (fF)
Standard Cells (original)	303.9	688.7	0.185
Standard Cells (improved)	38.5	93.8	0.302
Datapath Cells (original)	350.3	701.5	0.163
Datapath Cells (improved)	27.7	85.7	0.275

To further improve global extraction, several utilities were written which allow Dracula-generated capacitance values to be incorporated into the Cascade timing methodology. The main focus of these utilities is syntactically matching the capacitances in a Dracula extracted SPICE netlist with the corresponding nets in the Cascade database. Hash tables and linked-lists are used extensively to provide an efficient runtime. Used during the latter phase of the Aurora II design, these utilities were in part responsible for the close agreement (within 10%) between predicted clock frequency and that measured in testing. The use of Dracula capacitances should be reserved until the very last stage of timing analysis, since generating these capacitances requires a successful layout-versus-schematic (LVS) check. Typically, cell development and validation proceed in parallel with the other aspects of the design and are not completed until late in the design process.

5.4 Clock Phase Hazards

A two-phase level-sensitive non-overlapping clocking scheme was originally chosen for conservative reasons, since if circuits are properly designed this approach ensures functionality at some clock frequency. Some design errors can be worked around by adjusting the frequency and placement of clock edges on a tester in order to verify the functionality of a chip. On the other hand, a flip-flop based design requires much better analysis. A decade ago it was reasonable to assume that gate delay was much larger than interconnect

delay and the control of clock skew did not greatly impact the functionality of a design. Current process technologies support gate switching speeds of 100 to 200 ps and the component of overall path delay due to interconnect has increased greatly; for GaAs, average loaded gate delays of 100 to 150 ps mean that RC delay comprises 40% to 60% of the overall path delay. Consequently, much better analysis of clock skew is necessary to ensure proper functionality (clearly, skew also has an impact on performance). To illustrate this, consider Figure 5.2 which shows that if the clock is delayed to the second flip-flop due to a longer interconnect path, this flip-flop may incorrectly latch the data that has propagated through the first flip-flop.

The use of two phase clocking introduces a potential design hazard which can limit the frequency at which a chip will operate. Figure 5.3 shows a simple representative example, where signals latched in a previous phase are to be used to generate signals that are latched in the subsequent phase. This logic inadvertently uses inputs from both clock phases, resulting in a reduced active time for the Phi2 phase; it is constrained to be equal to the propagation delay through this logic block. An easily-implemented two-phase non-overlapping clock generator has an active period for each phase that is equal to 1/4 of the clock period; the 2 non-overlap regions account for the other 1/2 of the period. For a 4 ns clock (250 MHz), the active period would be 1 ns. However, the delay along a critical path may be almost twice this amount; the hazard means that the active time for the Phi2 clock will need to be long enough to accommodate this delay. Several of these errors were discovered in the Aurora II design when it was fabricated. More rigorous nomenclature might facilitate recognizing these cases, but many times the logic is complex and is derived from numerous

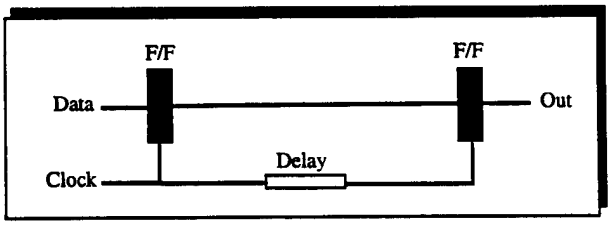


Figure 5.2 Clock Skew for A Flip-Flop Based Design

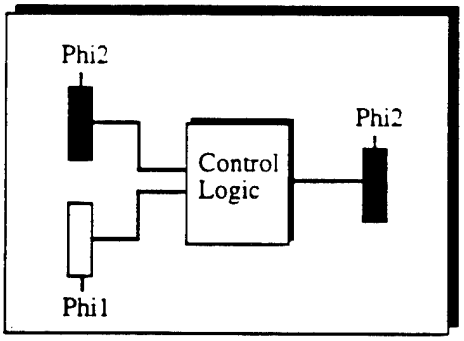


Figure 5.3 Clock Hazard for 2 Phase Design

intermediate signals. To ensure completeness, an automated check must be performed; to do the verification, a utility based on the delay-calculator traversal engine was written. Using the recursive algorithm described in Section 5.2, the utility visits every bit of every latch in the design only once. Whenever a latch is encountered, a second recursive routine is used to travel forward through the design in order to reach all gates driven by the latch. If a path terminates at another latch, information about this latch is added to a hash table. After traversing the entire design, the tool examines the hash table to identify any occurrences of two successive latches being driven by the same clock phase. For reasons which are discussed in Section 5.7, the runtime for this utility can be quite long. For example, the tool verified the FPU design in 12 hours, identifying 20 hazards. This utility is run only a few times near the end of the design cycle, and so a longer runtime is acceptable.

5.5 Clock Distribution Analysis

Level sensitive latches and phase borrowing (as discussed in Section 5.7) are somewhat more tolerant of clock skew, however accurate analysis of the clock distribution network is still critical for a high clock rate processor. An initial starting point for the distribution network was based on insight gained during completion of the Aurora II design. In the Aurora II CPU, both clock phases enter the chip from the same side and, although close in proximity, are separated by several ground pads. It became evident while testing the initial design that cross-talk can be significant if the clocks are assigned to adjacent pad locations. Both phases are routed to a central location on the chip where on-chip

clocks are generated and driven into a five-level distribution network. The non-overlapping clock phases distributed internally are generated from the external clocks, which are 90 degrees out of phase, using the following relationships:

$$Phi1 = \overline{\overline{clk1} + clk2}$$

$$Phi2 = \overline{clk1 + \overline{clk2}}$$

In addition, the sense amplifier-based register file requires a third phase, which is formed from an XOR of the two external clocks. The overall goal is to constrain latch-to-latch skew to be no more than 600 to 700 ps; an empirical rule limits fanout along each branch of the network to no more than 9. Each standard cell is assumed to provide a load of 2 minimum-sized enhancement transistors (15 microns of width), while datapath instances terminate in a column driver (50 microns of width). Very large buffers are used along the first levels of the distribution network. The last stage of the network ends at a local driver cell, which for datapath cells is physically located in the first row of a column. In order to keep polarity uniform throughout the chip, we added drivers to each standard-cell latch as well. A final post-processing step in the design methodology reduces the number of drivers by merging multiple latches so that they share a common driver.

Several utilities were written to help analyze the clock distribution network. The primary one, based on the delay-calculator traversal routine, generates a ready-to-run HSPICE netlist for each clock phase of the network. It first builds a data structure to represent the circuit, and then recursively moves outward from the top-level clock inputs. Each gate encountered along the way is added to a hash table; this proceeds until all latches that terminate the distribution network have been visited. A final routine reconstructs just the distribution network and creates a sensitized HSPICE netlist which contains capacitances, specific gate types, and fanout. As with other delay calculator-based utilities, Dracula generated capacitances can be incorporated into the output netlist. After HSPICE is run for both clock phases, several analysis scripts are used to provide different ways of analyzing the data. First, the transit time delays for both phases are sorted and plotted, as shown in Figure 5.4 for the FPU. A rough sense of skew in the design can be obtained by comparing

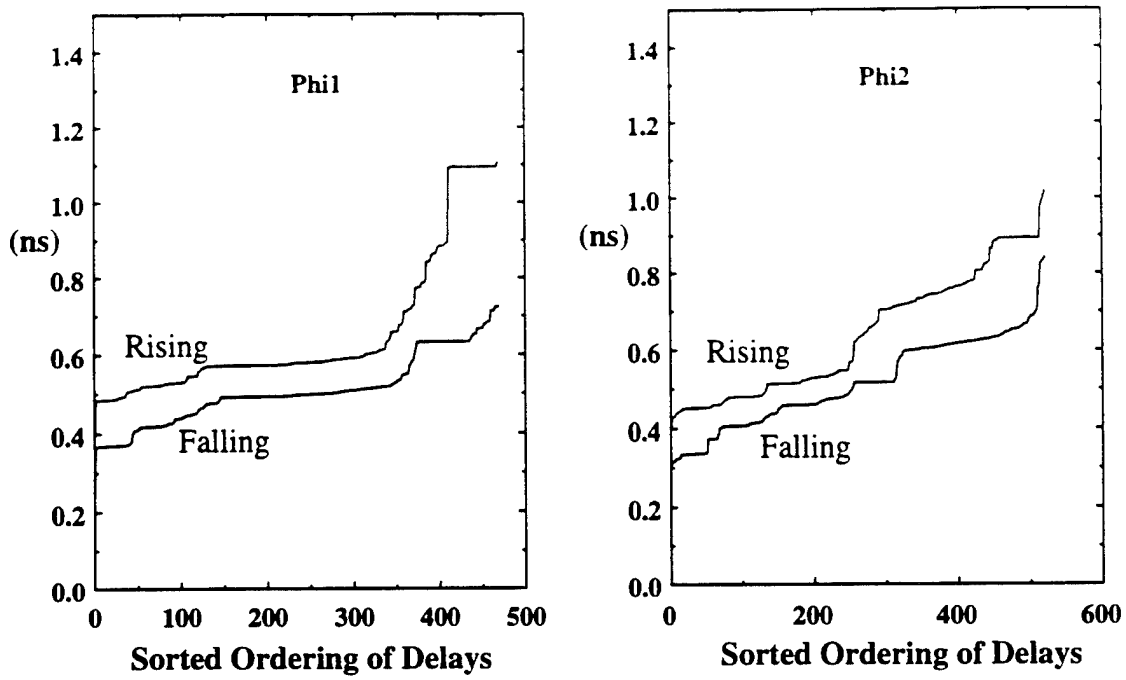


Figure 5.4 Sorted Clock Transit Times (No Resistances)

the range of values for these 2 graphs; a possible maximum value would 0.72 ns (1.1 ns of the Phi1 graph minus 0.38 ns of the Phi2 graph). This does not necessarily mean that 2 successive latches experience a skew of this magnitude. In order to obtain a more accurate report of latch-to-latch skew, a utility similar to the clock phase checking program was created. This program creates an ASCII database of all successive latch pairings; it is used in conjunction with the output from HSPICE to obtain a list of those latch pairs for which the skew exceeds a user-specified threshold. A final utility can be used to generate a 3-dimensional representation of clock transit time versus location on the chip. Figure 5.5 is such a plot for the Aurora II CPU.

Interconnect resistance along the distribution network can significantly impact clock skew; Figure 5.6 shows the sorted transit times for the Phi1 clock phase once resistances are added; it is evident that these resistances must be reduced. The problem is addressed by both increasing the size of the wires that are routed for the distribution network and by constraining this routing to use only the upper two interconnect layers, which have a lower resistivity. Both of these actions are initiated by adding attributes in Floorplanner to the nets in question. A script is used to identify problem nets, calculate the appropriate

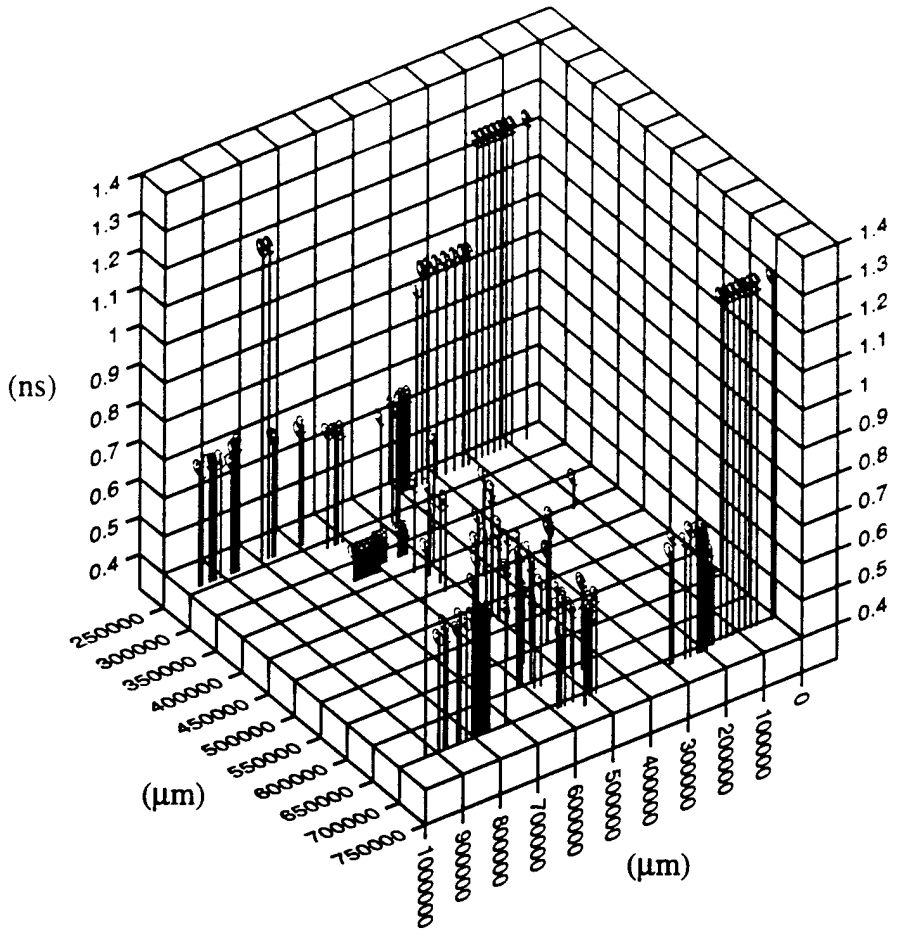


Figure 5.5 Clock Transit Time vs. Chip Location for Aurora II CPU

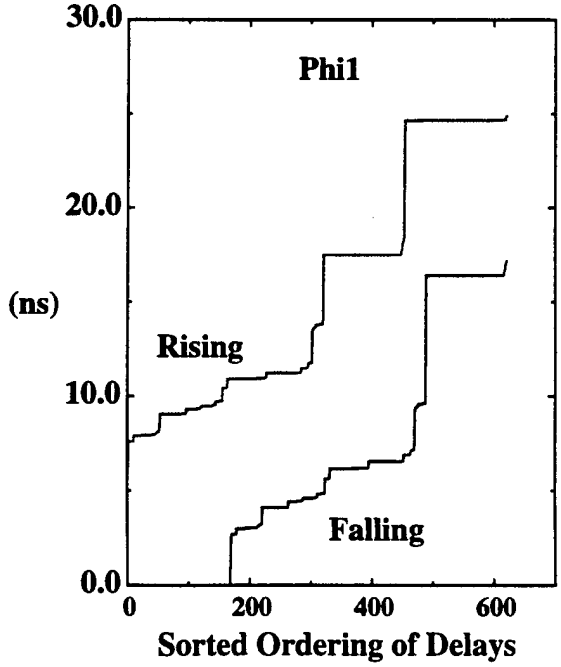


Figure 5.6 Sorted Clock Transit Times (With Initial Resistances)

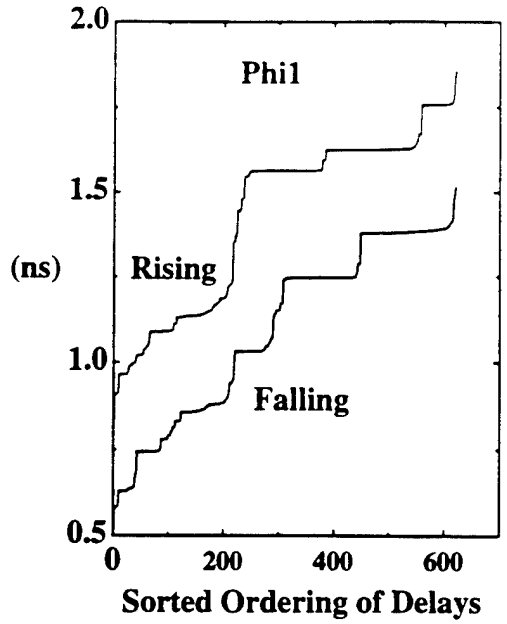


Figure 5.7 Sorted Clock Transit Times (With Final Resistances)

wire sizes, and generate a second script of attribute commands which can be invoked from within Floorplanner. When resistances are reduced to less than 50 ohms per net, the clock transit times are found to be reasonable, as shown in Figure 5.7. The analysis of resistance is discussed further in Section 5.6.

5.6 Resistance Extraction

As a quick and easy tool, a Perl script was written to extract resistances from a design. The runtime for the FPU is 2 hours, which is reasonable since this step is used primarily toward the end of the design cycle. This script utilizes a Cascade database-viewing tool (Proman) to find resistances that are local to all datapath partitions, and then proceeds to extract resistances for top level global nets. Since a net can be comprised of numerous branches, the resistance for the net is derived from the longest branch. The resistances and a summary of the percentage that each layer contributes to the average net-route are written to a text file, which is then used by a wire sizing script. This sizing script makes use of the following relationship:

$$Resistance = \frac{m1ratio \cdot m1shres \cdot m1pathlength}{Width} + \frac{m2ratio \cdot m2shres \cdot m2pathlength}{Width}$$

$$+ \frac{m3ratio \cdot m3shres \cdot m3pathlength}{Width}$$

The interconnect ratios (m1ratio, m2ratio, m3ratio) in this equation are obtained from the resistance extraction script and represent the percentage of a typical net that is on a given layer. Since it is important to minimize the resistance along the clock distribution network, these nets are routed only on the upper two interconnect layers, which have less resistance than the lowest routing layer (a factor of two lower for the top layer). Clock wires are sized by assuming a worst-case situation where all routing is done on the second layer.

Our current static timing analysis methodology does not consider the effect of resistance, which results in a degree of inaccuracy when analyzing critical paths. A macro-model approach for deriving RC delay has been developed but has not yet been implemented [Kayssi93b]. The current script-based approach should be rewritten in C to improve performance and more closely couple this step to delay calculation. In addition, the user might specify a delay threshold below which RC delay is ignored, in order to improve the runtime.

5.7 Determination of Gate Path Length

Improving the performance of a design is an iterative process, involving the following steps:

1. Identify all situations where the number of gates that lie along a path exceeds the allowable target for one clock phase. For a 4 ns clock period (250 MHz) and an average loaded gate delay of 100 ps, this target is set at 20 gates per clock phase.

2. Reduce the number of gates along a critical path, which can be done in several ways: a) reimplement a synthesized logic block by hand, b) factor out any late arriving signals, and c) retime logic by shifting it into the previous or next clock phase. The application of these approaches will often uncover additional critical paths and retiming may make paths critical which were previously acceptable. Consequently,

it is necessary to iterate over these first two steps.

3. Identify instances where capacitance, resistance, and/or fanout result in individual gate delays that exceed the goal of 100ps per gate and lead to path delays that are greater than the target delay. Approaches to reducing path delay include: a) replace DCFL gate instances with a buffered counterpart, b) change the size of the driving gate, c) reduce fanout by using multiple copies of the driving gate, d) reduce interconnect capacitance through better placement, routing, and cell design, e) reduce interconnect resistance by sizing wires and by constraining the layers used for routing.

The Cascade timing analyzer (Tactic) can be used to perform the analysis, but there are several reasons why this is not the best solution for the first 2 steps. First, Tactic presents paths in a graphical manner, which is beneficial for visually identifying components that comprise the overall path delay, but tends to be inefficient for summarizing just the gate depth along the path. A text-base report is more efficient at this point since it gives the user flexibility in sorting and searching through the list of paths. Second, support for a visual interface adds overhead to the runtime. For a complex design, it may be necessary to iterate many times over the first 2 steps so the total iteration cycle needs to be no more than two hours. In addition, at least one step performed by the current version of Tactic appears to be extremely time-consuming, requiring several days for a large design. Finally, there are several analysis features that are not supported by Tactic which can simplify the process of reducing logic depth; these will be discussed below.

In order to provide an alternative for identifying and resolving logic depth, a levelizer based on the delay-calculator traversal engine was written. This utility was originally designed to report all paths from inputs to outputs for synthesized logic blocks and was later extended to identify all paths between latches. In supporting the latter, it became evident that identifying all paths from the inputs to the outputs for designs with large amounts of connectivity can easily become a computationally intensive task. The traversal engine ensures that each bit of a latch is only encountered once and the original algorithm for the levelizer can be summarized by the following:

```
if (inst == latch) {
  start_inst = inst;
  find_paths(inst,1);
  }
find_paths (inst, level) {
  for (i=0; i<num_load_instances(inst); i++) {
      if (inst.successor_inst[i] == latch or inst.output[i] == primary_output)
        add_inst_to_successor_list(start_inst, inst.successor_inst[i], level);
      else find_paths(inst.successor_inst[i], level+1);
      }
  }
```

After the circuit has been traversed, a post-processing routine is invoked which sorts through the hash table created by add_inst_to_successor() to identify paths whose depth exceeds a level specified by the user. The result is a list of all possible paths between inputs, outputs, and latches. The requirement of traversing all inputs to all outputs can become quite costly for designs that exhibit a large degree of connectivity. Consider the heavily interconnected example of Figure 5.8. Assuming each gate in a column drives multiple gates in the subsequent column, the number of calls to find_paths() will be on the order of:

$$Calls = (\frac{Gates}{Column})^{NumColumns}$$

The Wallace array of the multiply unit is a good example of a highly interconnected circuit; the runtime for this design is approximately 12 hours. However, the runtime can be

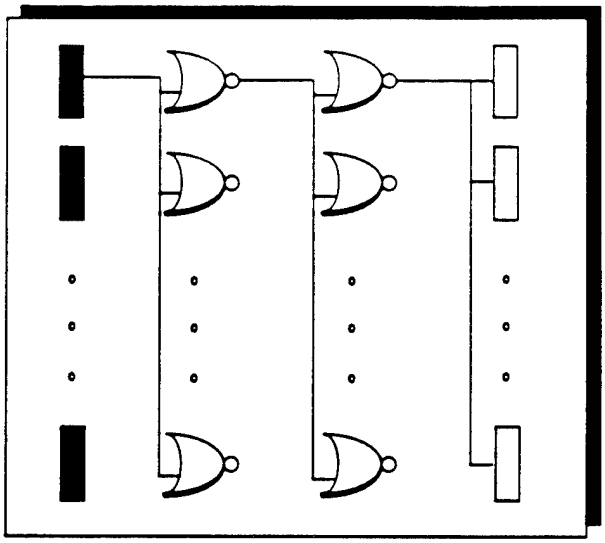


Figure 5.8 Run-time Increase For High Degree of Connectivity

dramatically reduced by constraining the algorithm to find only the worst case path between latches and inputs/outputs. Doing so simply requires keeping track of the worst level seen at any gate; this version of find_paths() can be summarized by the following:

```
find_paths (inst, level) {
  if (level > inst.level) inst.level = level;
  for (i=0; i<num_load_instances(inst); i++) {
      if (inst.successor_inst[i] == latch)
          add_inst_to_successor_list(start_inst, inst.successor_inst[i], level);
      else if ((level+1) > inst.successor_inst[i].level)
          find_paths(inst.successor_inst[i], level+1);
      }
  }
```

This approach prevents the explosive increase in calls to find_paths() by keeping track of only the worst-case path to each instance. Any attempt to continue expanding outward in a given direction will be inhibited if a previous traversal with a longer path length has already been encountered. The corresponding runtime for a design such as the multiply

unit is now on the order of an hour, a reduction in runtime of an order of magnitude.

Several additional features of the levelizer facilitate the process of reducing gate path length. The use of a two-phase latch-based clocking scheme allows the chance to borrow time from the clock phase which precedes a critical path. Since latches are transparent during the entire active time of a clock phase, it is possible that the combinational logic that drives the input to a latch may stabilize prior to the enabling edge of the clock. This stable signal is then immediately available to drive logic in the subsequent phase. As a result, a critical path is really comprised of the sum of the gate-depth for the logic that occurs in both clock phases. Although 20 gates per phase has been chosen as a design target, in selected cases this constraint can be relaxed if the worst-case path that drives the current critical path has fewer than 20 gates. It is important to control clock skew along these path-pairs, since a clock that arrives late to the first-phase latch will reduce the ability to borrow time from this phase. The levelizer generates a 3D histogram (Figure 5.9, the multiply unit) for which the x-axis is the level of the previous worst-case path, the y-axis is the level of the current maximum path, and the z-axis is the number of instances which have this previous-current pair. Figure 5.10 and Figure 5.11 show these plots for the overall FPU (excluding the functional units) and the add unit; note that the count (z-axis) is inhibited to make the plots more readable. These plots represent the results from numerous iterations over the first 2 steps described above. An additional option will print the worst-case path for the phase which

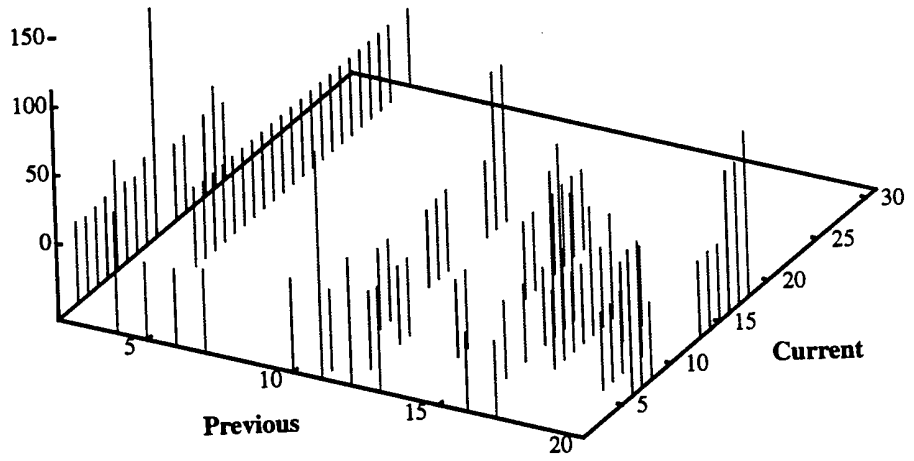


Figure 5.9 Multiply Unit Critical Paths of Current and Previous Phase

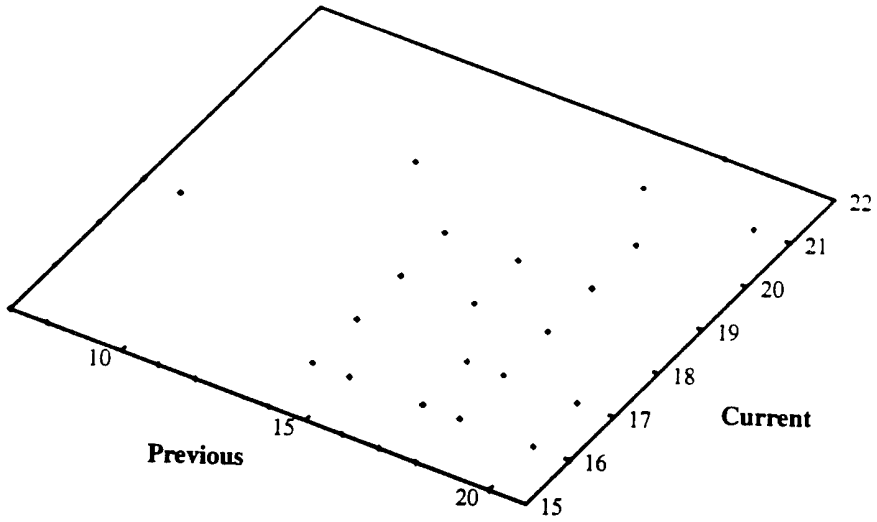


Figure 5.10 FPU Critical Paths of Current and Previous Phase (excl. FU's)

succeeds the current one; this information is useful when moving logic across latches while performing manual retiming. The levelizer also generates a list of the 50 instances that appear most often along all paths, allowing the user to focus on the most troublesome logic blocks.

5.8 Post-processing Optimization Utilities

Several additional functions have been incorporated into a single post-processing

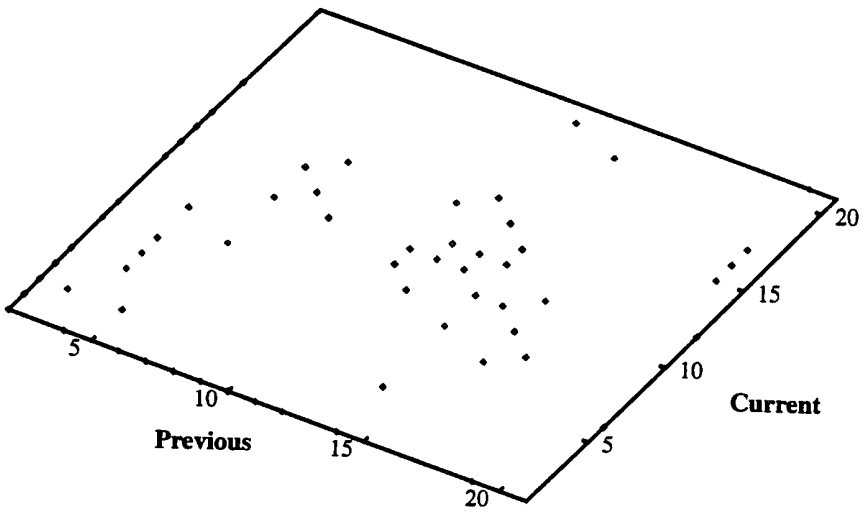


Figure 5.11 Add Unit Critical Paths of Current and Previous Phase

optimization utility, nicknamed the "gobbler" due to its ability to remove and transform logic instances. First, automatic buffer selection can be performed by running the stand-alone delay-calculator, which lists instances whose delay, output capacitance, or fanout exceed a user specified threshold. This text database is then used by the optimization tool to translate instances having large delays into buffered versions with the same functionality. Figure 5.12 compares delay, capacitance, and fanout for the FPU before and after buffer selection. Delays are improved significantly. Capacitance decreases slightly due to several layout optimizations that were performed in the second iteration. The large fanout points appear in part because a reset distribution tree had not yet been implemented; other large fanout instances correspond to the larger delay points. This comprehensive approach trades slightly larger area and power dissipation for computational efficiency. In other words, this approach is simple to implement but runs the risk of replacing DCFL gates with larger buffered versions for instances which have individually large delays but that do not extend appreciably the overall clock period.

A second function of the optimization utility involves automatically improving the gate-depth of logic blocks by searching for certain common logic sequences. For example, Figure 5.13 shows a sequence that occurs frequently within logic that has been synthesized by the Cascade utility Finesse. In this case, 2 levels of logic can be removed by merging the inputs of gate 1 and gate 3 by increasing the fanin for gate 3. Some other pattern matching optimizations which improve either logic depth or area due to fanin are:

1. 2 successive inverters can be removed altogether, assuming they are not needed for buffering reasons.

2. A constant input can be factored out of a gate, thereby reducing the fanin of the gate and/or propagating the output. For instance, a logic one on a NOR gate can be passed on to the successive gate as a logic zero. In turn, if the successive gate is a NOR function this logic zero can be factored out, allowing a reduction of the fanin of the NOR gate. These sequences can occur as a result of poor logic synthesis, or in redundancy that can occur at the interfaces between logic blocks.

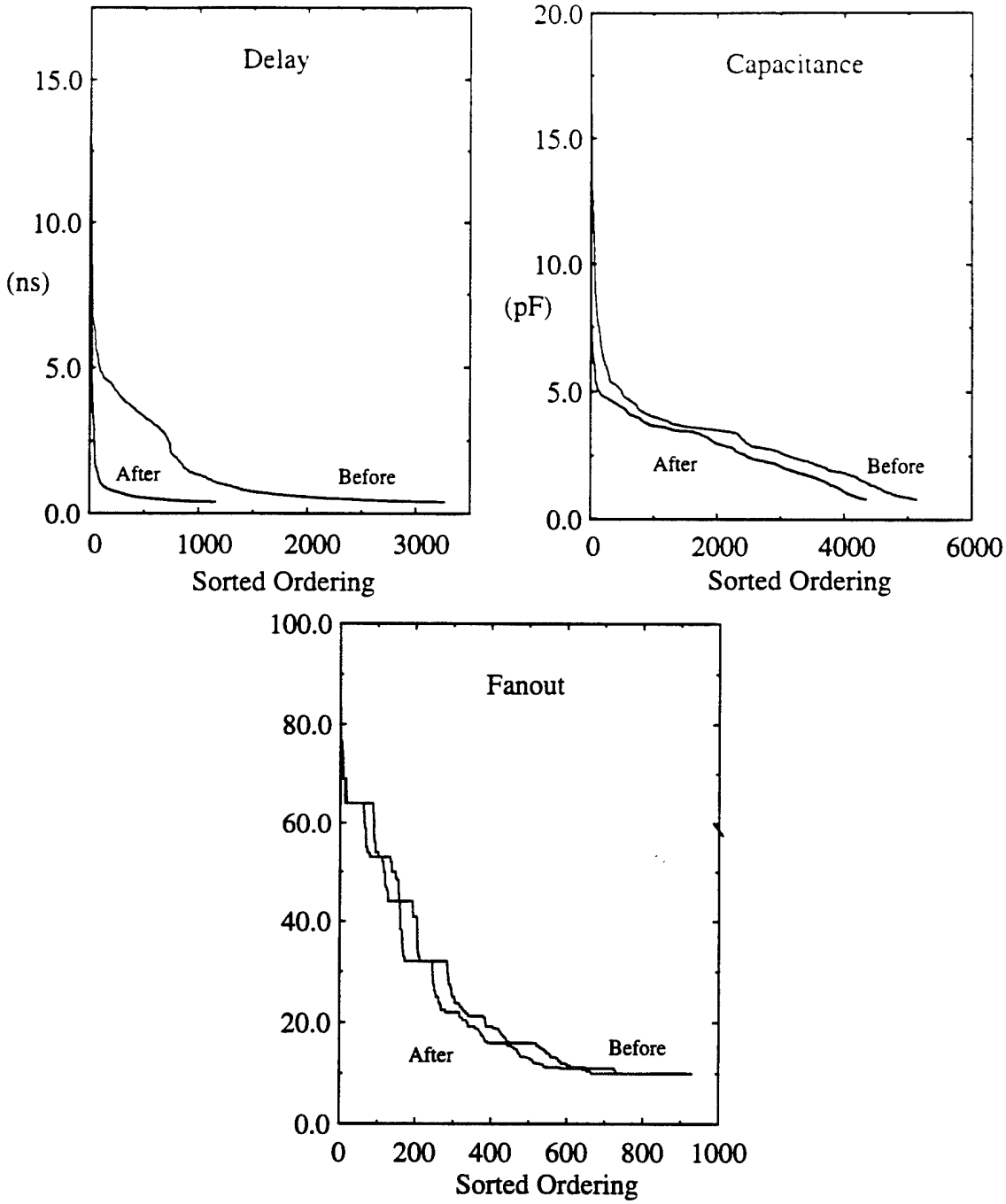


Figure 5.12 FPU Delay, Capacitance, Fanout Before and After Buffer Selection

3. A gate can be replaced by its dual, as in the case of a NOR gate, whose inputs are complemented, being replaced with an AND gate.

4. Gates which are driven by the same inputs can be merged into a single gate, assuming fanin constraints are not violated.

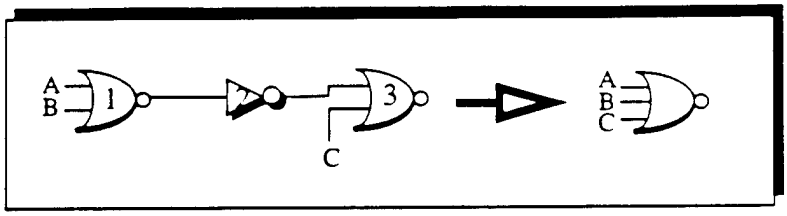


Figure 5.13 Pattern Matching Logic Optimization

Care must be taken to not inadvertently remove redundant, yet necessary, logic. The various stages of the clock distribution network form buffer-pair sequences which do not serve a functional purpose but are necessary for timing reasons. To avoid problems, this utility recognizes a "no_touch" property that can be attached to special instances that are to be excluded from optimization. The generation of a logic one in GaAs is another such example. A voltage of Vdd on the gate will forward bias the gate channel diode, will probably destroy the inverting transistor, and may result in an incorrect value for the output of the gate. A buffer whose input is tied to ground is typically used to generate a logic one. It is important that this buffer not be removed from the design during the propagation of constants; the "no_touch" property is used to ensure this.

A final function of the optimization program involves merging standard cell latch drivers. As was discussed earlier, a simpler design style with regard to clock polarity results if standard cells use clock buffers to match the polarity at the column drivers for datapath latches. This driver buffers the clock locally and decodes any select signals for merged logic-latch cells. Whereas a column driver is amortized across the many bits of a datapath column, it is costly to have a single driver for every standard cell latch. This tool therefore merges the drivers for multiple latches into just one instance, constrained by a user specified maximum fanout. For the FPU, this optimization removed approximately 900 latch drivers.

5.9 Miscellaneous Utilities

Several small tools were written to address issues such as beta ratio checking, power rail sizing, and determination of power dissipation. The beta-ratio for a DCFL gate sets both

the speed and noise margins, and a mistake in a cell can prevent proper functionality. To verify beta-ratios, a version of the delay-calculator traversal engine was used to ensure that all versions of all gate types in a design have been checked at least once.

Power rails for datapath cells need to be sized such that the voltage drop to any cell in the column is acceptable. A Perl script was written which will query the design database in order to identify for each cell the sum of the widths of all enhancement and depletion transistors that are connected to Vdd. Several empirical relationships for current as a function of transistor width, generated by iterative HSPICE runs, are used to estimate the current for each cell. This information is then used in conjunction with the following relationship in order to derive the width of the Vdd rail for each cell:

$$Width_{Vdd} = \frac{pitch \cdot Rsh \cdot N_{bits} \cdot I_{cell}}{IR_{max} \cdot DR \cdot N_{rails}}$$

$pitch$ = The maximum height of a datapath in microns; usually this is the maximum bit-width (32 or 64) times the effective cell pitch (60 to 70 microns)

R_{sh} = The sheet resistance for metal3, the layer used for routing Vdd

N_{bits} = The maximum bit-width for all instances of this cell

I_{cell} = The cell current derived from empirical relationships

IR_{max} = The maximum allowable voltage drop along the Vdd rails

DR = A derating factor that takes into account both the fact that a column is connected on the top and bottom to global Vdd and the fact that current decreases with distance down the column; generally a value of four is used

N_{rails} = The number of Vdd rails within a cell

This script also generates an estimate for total power. Since the empirical relationships do not consider the dynamic nature of power for buffer cells, this estimate should be

somewhat inaccurate. A more realistic value for power would be obtained by deriving an empirical relationship for dynamic buffer power as a function of interconnect, fanout, and frequency. The delay-calculator traversal engine could be used to calculate the interconnect and fanout on a per-instance basis.

5.10 Observations About Verification

Verification consumes an ever-increasing share of design time as processors become more complex. Both functionality and performance must be verified in a computer design before it is fabricated. Timing analysis has been discussed as a means of improving cycle time. However a design that is functionally correct may incorporate errors that limit performance by causing additional cycles to be executed. For instance, consider the head and tail pointers used to index into a reorder buffer. If these pointers are not advanced correctly, it is possible to operate in such a way that entries are not all allowed to be simultaneously active. Instructions will execute correctly, but more issue-stalls will occur since the reorder buffer will appear to be full more often than it should. Similarly, an inefficient implementation of a state machine might require unnecessary state transitions and add extra cycles of latency to an operation. Identifying such errors involves comparing cycle counts for a high-level architectural model with those of the actual structural implementation. A compact loop which increments a certain memory location some number of times can be used to detect such errors.

Among other issues, functional verification requires a choice of simulation environment. We have found that 60% to 70% of all bugs are identified using tests that are hand-generated by the designer. Since these tests tend to be fairly compact, an environment such as Verilog with a simulation speed of several cycles per second is appropriate. Random testing should find 20% to 35% of errors. Since many millions of instructions will need to be run for these tests, a compiled code simulator such as VCS, which offers a simulation speed on the order of 30 to 50 cycles per second, is recommended. Running verification in parallel on 10 machines for 4 months would allow approximately 5 billion cycles (a few billion instructions) to be run. Eventually, actual application- and OS-code needs to be ver-

ified; for this, a hardware emulation system would be appropriate. This approach offers simulation speeds between 500 Hertz and a few kilo-Hertz. While the use of compiled-code simulation can be fairly transparent to the user, hardware emulation tends to involve more man-power. At the present time, software tools that support hardware emulation are somewhat immature and inefficient, and it is not uncommon for small design changes to require turn-around times of several days.

Functional verification of the FPU depended heavily on randomly generated tests to exercise individual functional units, the full chip, and the FPU in the Aurora III system. Functional units were verified though the use of randomly generated operations, operands, and rounding modes. A specific computation was performed on both the compiled Verilog machine model and on an actual workstation. Data results and exceptions were compared and discrepancies were highlighted. Separate test modes were created for each of the functional units to allow errors found during random testing to be quickly rerun on an individual basis. In all, between 5 and 10 million operations were run successfully for all functional units. Verification of a commercial chip would have to be more extensive, with many billions of operations being simulated. The level of verification performed was considered appropriate for the resources of an academic effort. Hardware emulation would provide the best support for more extensive validation.

As mentioned, chip-level verification was done in a random test generation environment. A behavioral model for the IPU was created to feed the FPU with instructions and data. Tests are generated from 35 primitive instruction sequences. In general, there is one primitive for each piece of functionality in the FPU, such as add, multiply, or divide instructions. A typical sequence will execute a floating-point instruction and compare the result to the known correct value, making the tests self-checking. The simulation environment loads several queues at the start of a test with the correct outcome for floating-point comparisons and stores. The primitives are randomly tiled together and a variable number of NOPS are added. The program that results is run on the compiled-code version of the simulator. Instead of requiring this FPU test environment to read an actual program binary, pseudo instructions are used and the resulting program consists of ASCII hex values. This approach

to testing offers several advantages. First, because the tests are self-checking, there is no need to compare results against a separate high-level behavioral model of the design. With limited resources, it is difficult to support efforts to design separate trace-driven, behavioral, and structural representations of a design. Second, each random test is fairly short, on the order of several hundred instructions, making it easy to locate errors. Even short programs typically involve several thousand instructions, making it difficult to identify errors. Finally, each test is self-contained, which allows verification to be infinitely parallelizable. The only limitations are the number of available workstations on a network and the number of licensed copies of the simulation environment. Starting from the moment random verification commenced, over 200 million floating-point instructions were run successfully.

Several observations about random verification were made in our experience with the FPU. For example, if the primitives used are not short enough, the full benefit of tiling the primitives is not realized and instead the only benefit will be based on the number of permutations that are derived from inserting noops. In our initial tests, each primitive started with several load instructions to initialize the operands. This meant that in the tests, each primitive was succeeded by a load instruction, rather than all possible permutations. A constant header block was therefore added to each random test. Within this header, all operands used for computations and all results used for self-checking verification were loaded into the first 23 registers of the floating-point register file. The remaining 9 registers were used for dynamic results. At first, a floating-point compare instruction always ended each primitive and tiling only resulted in testing pairings of compare instructions with other instructions. Again, not all permutations were being allowed. The solution made use of the following sequence:

1. Buffer the result of the last compare in a primitive.
2. If the destination register of the first instruction of the next primitive is the source of this compare, issue the compare and then the new instruction, otherwise issue the new instruction followed by the compare. To reduce the occurrence of the former condition, the source operands of the compare are always registers Fa and Fb, and the destination of the initial instruction is always register Fc.

The last form of verification was performed at the system level, where the IPU and FPU are connected. The primitives used in this simulation exercise the same functionality as in the chip-level tests, but now consist of actual MIPS instructions. This environment serves to more comprehensively test the actual interface between the two chips. Since the models for both chips are fully realized, all functionality is simulated, including cache misses. As a result, instruction and data misses lower the throughput of floating-point instructions.

Random verification offers an efficient means of resolving the majority of bugs in a design. However, it fails to provide complete coverage. This is due to several reasons. First, a designer will still tend to include assumptions in the infrastructure for random testing that leave certain areas untested. For example, using random testing for the verification of a functional unit needs to consider different regions of operation. The add unit is comprised of different pieces of logic that are each exercised depending on whether an alignment of one, no alignment, or an alignment of greater than one is necessary. Failure to force operands to fall equally within each of these regimes would result in inadequate coverage.

Random testing makes it difficult to discover unusual boundary cases that are dependent on the occurrence of a sequence of events, such as an instruction miss which occurs for a branch, followed by a page-fault in the delay slot, and while this fault is being serviced, an interrupt. A logic error might result in the wrong program-counter being selected by the time the delay-slot instruction is finally executed. Such a sequence of events can be quite difficult to generate, and even if several billion instructions are executed, the error may never be encountered. As designs become more complex, interest has developed in pursuing more formal approaches to verification.

5.11 Future Work: A Methodology for Automatic Logic Optimization

This section will summarize several ideas for an automated approach to performing timing analysis and logic optimization; many of the individual steps in this methodology have been mentioned in the preceding sections. During the course of completing the timing

phase of the Aurora III IPU and FPU designs, it became clear that much effort was being spent on the same set of tasks, each of which is well defined and reasonably straight-forward to implement automatically. Part of the challenge in implementing an automated version of this methodology centers on how to iterate between these steps. A summary of the methodology is:

1. Levelization is performed in order to identify paths which exceed the targeted gate depth. Logic can be optimized in the following 3 ways:

   a. Optimal synthesis for logic which has fewer than 10 inputs/outputs, perhaps using some sort of exhaustive branch-and-bound algorithm. The majority of logic within a design is in blocks having few inputs and producing few outputs (refer to Table 5.2). Current synthesis tools often produce results which are far from optimal; this problem is made worse by the limited library of gates available in GaAs. Table 5.3 compares several logic blocks that were synthesized using GaAs and CMOS cell libraries, where the latter has access to more complex gate structures. The GaAs implementation often has many more levels than the CMOS version, and even for CMOS, the result is often several levels away from being optimal. The poor results for GaAs appear to be due to the technology mapping phase of the synthesis process. Similar results have been found using MIS, another synthesis tool. The need to redo large amounts of logic by hand is time consuming, and the discovery of a bug in the definition of the logic may mean that the same logic must be regenerated several times. Whereas, the original behavioral description for the logic may be easy to understand, the subsequent structural implementation may be quite difficult to follow.

   b. Factor out late-arriving signals through the use of a mux-reduction technique. Figure 5.14 demonstrates this idea. Signal A, which might be the carry-out of an adder, does not arrive until late in the current phase. However, the original synthesized logic is not aware of this fact and signal A is factored into the first level of the logic block. To solve this problem, two separate descriptions for the logic are used; one assumes input A is a zero and the other assumes input A is a one. The output from

Table 5.2 Average Inputs per Output for Control Logic

Control Block	Avg Inputs per Output	Max Inputs per Output	Total Outputs in Block
convertcontrol	4.524	9	21
iqcontrol	9.600	18	25
robcontrol	3.915	19	117
elogic	3.600	5	15
lqtagmemcontrol	4.000	7	24
sbcontrol	2.000	3	288
fpucontrol	15.840	77	169
precisetagmemcontrol	4.286	9	42
predecodelogicfpnew	8.840	17	25
stickybit	6.059	7	51
grsnew	7.857	12	35

Table 5.3 Gate-Depth for CMOS versus GaAs Logic Synthesis

Control Block	No. Gates (CMOS / GaAs)	No. Outputs	Avg Difference in Gate-Depth	Max Difference in Gate-Depth	Avg Gate-Depth (CMOS / GaAs)	Max Gate-Depth (CMOS / GaAs)
convertcontrol	68 / 102	21	0.476	3	3.476 / 3.952	6 / 8
iqcontrol	77 / 140	25	1.080	4	7.400 / 8.480	11 / 13
robcontrol	217 / 384	117	0.333	2	3.282 / 3.615	8 / 8
elogic	20 / 50	15	1.733	2	3.333 / 5.067	4 / 6
lqtagmemcontrol	41 / 77	24	1.250	2	3.000 / 4.250	5 / 6
sbcontrol	444 / 568	288	0.000	0	1.889 / 1.889	2 / 2
fpucontrol	757 / 1312	169	0.420	4	6.083 / 6.503	15 / 15
precisetagmemcontrol	78 / 128	42	1.429	2	3.000 / 4.429	6 / 8
predecodelogicfpnew	236 / 395	25	1.440	4	6.640 / 8.080	13 / 12
stickybit	133 / 231	51	0.353	2	5.529 / 5.882	10 / 10
grsnew	150 / 269	35	1.857	5	6.429 / 8.286	9 / 12

each description is fed into a multiplexor, whose select line is simply the late arriving signal. In this way, only 3 gate levels (1 for decoding, 2 for the multiplexor) are added to the path-depth of the logic that generates signal A. Synthesis is still used to generate the A=0 and A=1 signals, and because much of the logic is shared between these signals there is only a slight increase in instance count compared to the original version. Doing this transformation by hand is tedious and makes the resulting description more difficult to interpret.

c. Utilize a post-processing optimization phase in order to pattern match commonly occurring logic sequences. Propagation of constants, redundant logic at the interfaces between logic blocks, and poor synthesis can all be addressed in this way.

2. Repartition logic across latch boundaries. This will involve moving small pieces of logic to either the previous or next clock phase. This step might operate directly on the design database or generate appropriate Verilog code that can be integrated into the top level design. In either case, the user should be made aware of the changes being made.

3. Perform the closely-coupled steps of parasitic extraction, delay calculation, static timing analysis, and buffer sizing/selection. This stage analyzes the delay of critical paths and iterates primarily by choosing to size an existing driver or to select a buffered gate type for those instances that experience a long delay due to fanout or interconnect parasitics. Wires along critical paths which have large resistances

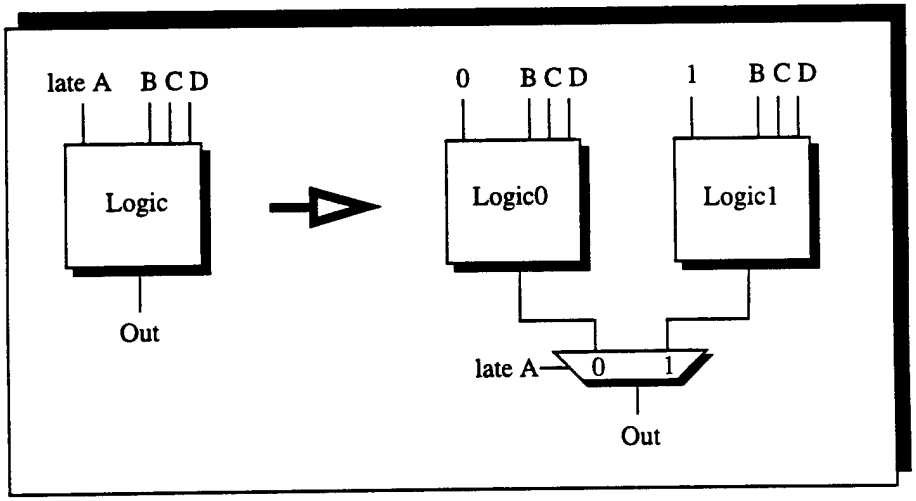


Figure 5.14 Factoring Late Arriving Signals Via Mux-Reduction

should be resized automatically, and this RC delay information should be included in path delays.

Automation of this sequence of steps will save much time on tasks which are tedious to do by hand. Since verification and performance optimization often occur in parallel, more time can be lost if bugs require repetition of some of the steps. Further, doing these steps by hand makes the high-level Verilog description of a design more difficult to interpret.

CHAPTER 6
Conclusion

This chapter summarizes the work and contributions presented in this thesis and discusses several related future areas of research.

6.1 Summary of GaAs Technology

Many of the design constraints for GaAs DCFL, including small noise margins, low threshold voltages, sensitivity to voltage drops along ground distribution, and over-driving of inputs along highly capacitive nets, are a result of the Schottky diode gate of MESFET transistors. We addressed the over-driving problem with the use of a feedback buffer, which provides a large dynamic current for charging a wire and a smaller static current for dc operation. The benefit of fast gate switching speeds in DCFL tends to be offset by greater gate-depth that results from a NOR-NOR logic topology. Transistor source resistance is large enough to limit the use of both stacked-transistor gates and pass gates, especially in light of small noise margins. Leakage currents are several orders of magnitude larger than for silicon devices and constrain the maximum fanin of a DCFL gate to four inputs. In addition, this issue affects the density of SRAM components by limiting the amortization of sense amplifiers across rows of an array. The use of dynamic logic is impractical due to the current that flows into the gate of a MESFET transistor. Gate current also limits the degree of fanout that can be supported by either DCFL or buffered gates. In current GaAs processes, interconnect has larger dimensions than in current CMOS processes. The importance of improving metal technology in GaAs was illustrated by contrasting the expected improvement to that of improving gate delays.

The current performance gains of GaAs DCFL process technology are not signifi-

cant enough to compel a current CMOS design effort to be retargeted for GaAs. Integration levels are lower than CMOS by a factor of 5 to 10. Much of this is due to the characteristics of ratioed logic, large leakage currents, and less efficient interconnect pitches, rather than to a difference in minimum transistor feature size. In terms of operating frequency, GaAs does offer faster loaded gate speeds, but this benefit is tempered by DCFL providing a smaller range of circuit design options and by needing longer gate-depths to produce the same functionality. Power dissipation at high frequencies has often been claimed as an advantage of GaAs. The static relationship of power dissipation in GaAs is contrasted with the strong dynamic component in CMOS and suggests a cross-over point for frequency at which GaAs becomes more power efficient. However, several trends in CMOS design have weakened this argument. Power supply voltages for CMOS have dropped from 5 volts to 3.3 volts and lower, which has raised the cross-over point. More importantly, there has been an effort to reduce power by turning off the clock to logic blocks which are not being actively utilized. Consequently, logic transitions and resulting dynamic power dissipation are reduced. This approach is not nearly as effective for GaAs, in light of the strong static component to power dissipation. Complementary GaAs (C-GaAs) offers the potential of addressing some of these issues, but with the caveat of being unproven at the present time. C-GaAs will support limited stacked transistor structures, which should reduce gate-depth along critical paths. Power dissipation for C-GaAs circuits should be lower since there is no dc path between power and ground. Current will still flow into the gate of a transistor and contribute a static component to power, but this is substantially less than for DCFL.

It seems unlikely that in the short term GaAs DCFL will become a significant alternative to CMOS for large VLSI designs. Consider Table 6.1, which shows a projection for DCFL and C-GaAs. In addition to a process improvement of 30% for loaded gate delay, the DCFL projection assumes that industrial-level resources are applied to the project; these resources would result in both several gate-levels being removed from all critical paths and an increase in density that translates into a 20% gain in clock frequency. For C-GaAs, a more deeply pipelined machine might achieve a logic-depth along critical paths of 15 gates per clock cycle. However, a more deeply pipelined design can suffer worse memory system

Table 6.1 Performance Projections for GaAs and C-GaAs

Technology	Avg Loaded Gate Delay (ps)	Gate-depth per Clock Cycle	Clock Frequency (MHz)
DCFL (1994)	100	40	250
DCFL (1995)	80	40	310
DCFL (1997)	50	36	500
C-GaAs (1997) Conservative	100	22	450
C-GaAs (1997) Optimistic #1	70	22	650
C-GaAs (1997) Optimistic #2	70	15	952

latency, will require more latches, which may constrain clock frequency, and will be more sensitive to miss-predicted branches. C-GaAs may provide a level of density for SRAM components that is appropriate to support a combination of on-chip caches, register blocks, and branch-prediction techniques used to reduce miss-predicted branches. A recent 1μm C-GaAs 4K-bit SRAM provides a density of 11,600 transistors per mm^2, an access time of 5.3ns, and a power dissipation of 16.2mW [Hallmark94]. In general, a target logic-depth of 15 gates per cycle will be a challenging goal for either an academic or industrial design effort. In comparison, CMOS processors have currently been demonstrated to operate at clock frequencies in the range of 200MHz to 300MHz; at the traditional growth rate of 50% per year, it is reasonable to expect the introduction within the next two years of CMOS processors with clock frequencies greater than 600MHz. The basic limitations just discussed for GaAs may not be due to technical reasons as much as to economic ones. In the absence of volume commercial products fabricated in GaAs, there might not be sufficient financial resources to advance the technology at a pace comparable to that of CMOS. An alternative technology probably needs to offer at least twice the performance in order to attract designers away from the large infrastructure and expertise currently available in silicon technology.

6.2 Summary of FPU Architectural Issues

The architectural study is based on a trace-driven simulator which was extended to describe the Aurora III integer and floating-point architecture. The simulator produces performance metrics, including average instruction latencies, dynamic instruction frequencies, basic block size, bus utilization, average degree of issue, sizes for different resources, stall sources, and cycles-per-instruction. Simulations were run using traces from the SPECfp92 benchmarks for a wide variety of architectural features.

The first of these experiments examined three issue policies for floating-point instructions, IOIO, IOOO, OOOO (the first and second pairs of characters mean in-order or out-of-order for issue and completion of instructions). A number of conclusions were drawn about the resource cost and performance benefit of each. The first policy is the simplest and achieves the worst CPI, although it does support a faster clock frequency by eliminating the need for a reorder buffer and by simplifying issue logic. The second policy, which utilizes more parallelism by allowing results to complete out of order, offers approximately 12% better SPECfp92 performance for a moderate increase in resources and design complexity. Dual-transfer and dual-issue of floating-point instructions were found to offer a combined 15% improvement in CPI versus a single issue approach. The third policy, OOOO, attempts to increase look-ahead capability by moving a data-dependent instruction past the decode phase, into an instruction window which resides between the decode and execute units. Ideally, this should allow a greater opportunity to find instructions without dependencies. However, the realized performance gains are small (a few percent), due primarily to an increased utilization of the reorder buffer. While most instruction types produce a result which can be used immediately upon writing the reorder buffer, floating-point comparison and store instructions must wait until the corresponding entry reaches the head of the reorder buffer in order to ensure precise handling of memory exceptions. Most of these synchronization stalls are due to branch-on-compare sequences. An OOOO policy results in more instructions being active, and a corresponding increase in the number of reorder buffer entries that precede a floating-point compare, which in turn reduces the intended

benefit of this issue policy. Further, the resources required by an OOOO policy are quite substantial; two reservation station entries per functional unit can add 25% to overall chip area. In other words, the additional resources needed to implement OOOO are equivalent to the area difference between a 2-cycle pipelined multiply unit and a 5-cycle iterative unit. The performance difference for this multiply unit trade-off is 10%, which is an improvement not matched by switching from IOIO to OOOO. Several additional aspects of an out-of-order issue policy are discussed, including design complexity and approaches for selecting instructions to be issued from a reservation station. All initial simulation experiments were run with large resources in order to identify an upper bound for performance. Information from these runs was used to choose a more reasonable allocation of resources which results in only a small degradation in performance (a few percent) from the ideal case of unlimited resources.

Although the FPU with the chosen issue policy (IOOO) and configuration generates few stall cycles due to the sizes of various resources, the other major stall source, branch-on-FPU comparisons, was found to cause a significant number of stalls on some benchmarks. A set of instruction sequences, consisting of a floating-point load followed by a compare and then a branch which depends on the result of the compare, was found to be common among these programs. A number of approaches were discussed for reducing the large latency associated with the compare instruction, including: moving ahead in time the transfer point for floating-point instructions, improving the primary memory system, and reordering code for these sequences in order to place more useful unrelated work between the load, compare, and branch instructions. All of these issues were investigated in the context of both single- and dual-issue designs. The best design point that resulted would reduce compare latency to simply the time needed to perform the comparison and transmit the result back to the IPU; the overall performance gain was approximately 10%, with some individual programs improving by as much as 23%.

Integration levels are low in GaAs and several techniques for improving memory system performance were discussed, including both greater bandwidth through the support of double-word load and store instructions, and data prefetching. An optimistic upper

bound for performance improvement of the former was found to be 10%, whereas the latter results in an overall reduction in CPI of 15% and improvement for individual benchmarks of as much as 60%.

A number of resource allocation issues were examined. First, performance was compared to resource requirements for functional units of various latencies. For example, a 2-cycle add unit offers only a 2% improvement in CPI versus a 3-cycle design, but at the expense of a 20% increase in area. However, some trade-offs are not as clear-cut, such as the multiply unit decision mentioned above. A 2-cycle pipelined multiply unit occupies twice the area of a 5-cycle iterative version, but achieves a 10% reduction in CPI. Ultimately, integration constraints prompted the selection of the slower, smaller multiply unit. A different solution might be reached in a CMOS design. Second, the allocation of transistors was examined in the context of queue and reorder buffer entries. Queue entries are fairly inexpensive, while reorder buffer entries are more costly since they are wider and require more read and write ports. Two to three entries are required for the load and store queues, which corresponds to the observation that most applications use the double precision format, requiring two loads or stores per operand. Finally, the area implications for each of the three issue policies were discussed. As mentioned, in-order issue and completion is the simplest, while out-of-order issue and completion requires substantially more storage space than the other policies.

Instruction and data queues allow the integer unit to slip ahead of the FPU, since the IPU does not necessarily need to stall as a result of floating-point data dependencies or resource conflicts. Decoupling queues also serve to hide latency caused by chip crossings and can lessen the impact of having different clock frequencies for the IPU and FPU, a situation that can result from differences in the width of datapaths. However, the use of queues does make support of precise exceptions more difficult. Characteristics that are unique to both memory and floating-point computational exceptions were discussed and several approaches are presented for handling these issues which do not have a significant impact on either chip area or performance.

In order to reduce pin count, the existing primary data cache busses were also used

for transferring instructions and data between the IPU and FPU. Up to two floating-point instructions can be transferred per cycle, which is the bandwidth needed to support dual issue of instructions in the FPU. Load data can originate from several sources, including the primary data cache, the write cache, the prefetch unit, or the secondary memory system. Floating-point store instructions are somewhat more complex than their integer counterparts, primarily because the data to be stored arrives sometime after the corresponding instruction. Several alternatives were proposed for handling both loads and stores. Altogether, these cases have been implemented in efficient manner which has little impact on performance and which requires only a small number of additional I/O pins.

Achieving high performance under all conditions in a computer having integer and floating-point capability requires attention to a number of details. An example of this covered in the dissertation is the use of result busses. As more parallelism of instruction execution occurs in the FPU, more demand is placed on the bus that is used to write results from an execution unit to the reorder buffer. The choice of using two result busses correlates with the average degree of issue of 1.3 instructions per cycle. The benefit of providing hardware support for square root is also analyzed. This operation is used mostly by multimedia applications, in which it is used to translate graphic-based objects. A dedicated square root instruction can improve the performance of the one SPECfp92 benchmark that relies on this operation by 50%, and overall across all benchmarks by 7% to 9%. Another trade-off which was analyzed was floating-point division, which can be implemented by reciprocal algorithms which use the multiply unit. However, the impact on performance in doing so is quite large, since frequently occurring multiply instructions cannot issue if a division operation is outstanding. The effect of miss-predicted branches on floating-point code was also considered. Because the basic block size for floating-point code is larger than that of integer code, there is less opportunity for branch prediction to improve performance. The static prediction policy used by the Aurora III architecture degrades CPI on floating-point code by only about 4%, compared to a perfect prediction policy; integer code can suffer a much larger penalty, on the order of a 30% degradation in CPI. Finally, simulation accuracy and run-time speed for trace-driven simulation is examined. Several benchmarks

are run for both the first 50 million and the first billion instructions and several metrics are used to compare the difference in results. While most benchmarks experience only a few percent change in CPI, one program (spice2g6) does see a 15% difference. Consequently, sampling should be used for reasons of both accuracy and simulation speed (but was not used in the Aurora III simulator due to time constraints for implementation and verification).

The architectural investigation was concluded by summarizing the current state of microprocessor performance, through the use of SPEC ratings. Five versions of the Aurora III FPU design were evaluated, including a 250MHz baseline. One of these designs emphasized the extraction of instruction-level parallelism through greater complexity while another focused on a simpler design which runs at a faster clock frequency. The other versions were projections of the baseline in light of expected technology improvements. The process technology that supports the FPU design has not changed in over two years. However, a newer version will soon be available and should decrease the average loaded gate delay by 10% to 40%. The final Aurora III design achieves a SPEC rating of 300, which compares favorably to the highest performance processors currently available (SGI TFP = 310, IBM/Motorola PowerPC 620 = 300, IBM Power2 = 274). Overall, improvements in clock frequency have a greater impact on improving floating-point performance than do architectural or algorithmic improvements. A number of variables which are not reflected in these SPEC ratings are also mentioned, including better code reordering, utilization of a larger register file, support for double-word loads and stores, and support for a hardware square root instruction. Together, these features might conservatively improve performance by an additional 20% to 30%, resulting in a SPEC rating in excess of 400.

While many of the architectural trade-offs presented are certain to be familiar internally to industry development efforts, this dissertation represents a unique, comprehensive, and accessible summary of important issues for supporting high-performance floating-point execution.

6.3 Floating-Point Implementation Issues

The results of simulation studies were applied to the design of an FPU for the Aurora III system in a GaAs DCFL technology. Adders of various sizes are basic components used in all of the functional units, so several alternatives were evaluated. A carry-skip design was not chosen since the small fanin that is a characteristic of GaAs increases the number of gate levels needed for the skip logic. Similarly, a group-4 carry look-ahead adder requires several additional gate-levels compared to the Ling-modified carry-select approach that was selected. Much of the motivation for the functional unit designs originated with work done elsewhere, but has been extended in order to accommodate differences that result from using GaAs. Also, a number of corrections to the original references were discovered during the verification of these units. More than 5 million random test vectors were performed for all computational units. New implementations for leading-one prediction and rounding logic in the add unit were presented. The conversion unit that is described is an original design that can generate any of 6 conversion operations with a latency of 2 cycles. The dissertation also examines a general set of implementation issues, including approaches for supporting precise exceptions without incurring a performance penalty, ways of handling floating-point loads and stores, the use of predecoding to reduce critical path depth, and design-for-test features.

6.4 CAD Support for High Performance VLSI Designs

The dissertation discusses a number of analysis tools that have been developed to provide feedback about timing and functionality. Many of these utilities are based on a delay calculation network traversal routine obtained from Cascade Design Automation. Initially, this delay calculator was updated to utilize a macro-model approach developed by another student in our research group. Several utilities were developed to aid verification of this new delay-calculator, the predictions of which have been found to be within 10% of HSPICE generated delays for all cases and within 7% for more than 60% of cases. The derivation of delays also depends on accurate parasitic extraction; several utilities were written

to address accuracy limitations in the physical design system that we used. These include the capability to incorporate capacitances obtained from commercial extraction tools into the delay calculation routines and the ability to analyze and adjust interconnect resistance through wire sizing and specification of which layers are be used during routing.

The first utility based on the delay traversal routine addresses a certain type of timing hazard that can result from a two phase level-sensitive clocking scheme and which can limit the frequency at which a chip will operate. More rigorous nomenclature can aid a designer in recognizing these cases, but this is often difficult because the logic can be quite complex. To ensure completeness, this utility provides an automated approach for identifying these errors.

Several programs were written to help analyze the clock distribution networks of the Aurora III chips. The primary one is also based on the delay traversal routines and generates a ready-to-run HSPICE netlist for each clock phase of the design. As with other delay-calculator based routines, capacitances generated outside of the Cascade environment (via Dracula or Mentor Graphics) can be incorporated into the output netlist. After HSPICE is run for both clock phases, several analysis scripts are used to provide different ways of analyzing the data, including sorted 2D plots of skew, 3D plots of clock transit times versus chip location, and a textual listing of latch-to-latch skew.

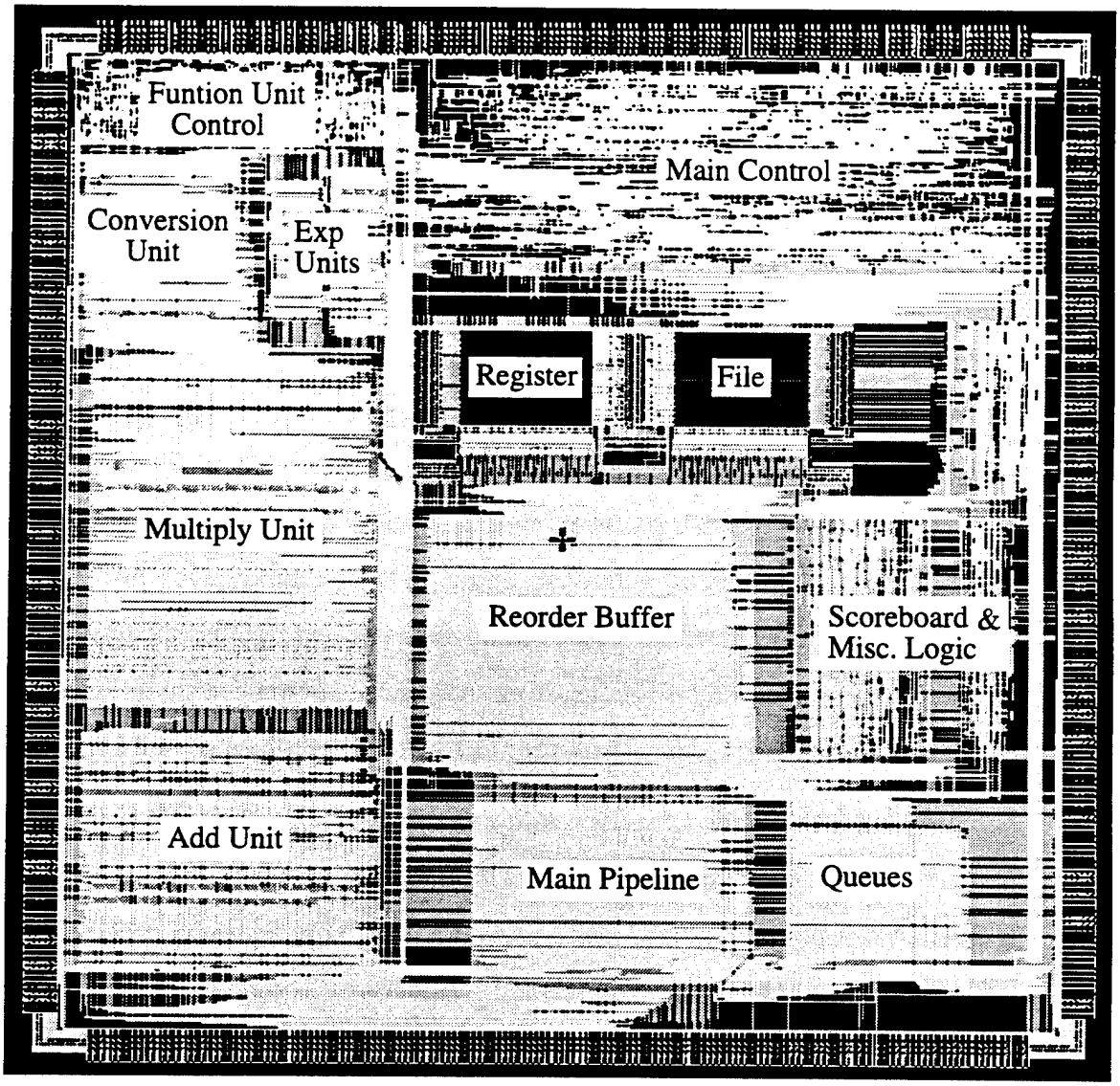
To complement the Cascade timing analyzer for the analysis of logic depth, a levelizer based on the delay traversal routine was written. The use of a two phase latch-based clocking scheme allows borrowing time from the clock phase which precedes a critical path. The levelizer generates 3D histograms for which the x-axis is the level of the previous worst-case path, the y-axis is the level of the current maximum path, and the z-axis represents the number of instances which have this previous-current pair. An additional option will print the worst-case path for the phase which succeeds the current one; this is beneficial when moving logic across latches while performing manual retiming. The levelizer also generates a list of the 50 instances that appear the most frequently along all paths, allowing the user to focus on the most troublesome logic blocks.

Several additional functions were added to an existing post-processing optimization utility obtained from Cascade. First, automatic buffer selection can be performed by running a stand-alone version of the delay-calculator. The resulting text database of delays, capacitance, and fanout is then utilized by the optimizer to translate instances with large delays into buffered versions which have the same functionality. Second, the optimizer improves gate-depth by searching for certain common logic sequences. This pattern-matching approach recognizes redundant logic that can occur between the interface of two logic blocks. This utility also merges the drivers for multiple latches into just one instance, constrained by a user-specified maximum fanout. In the FPU, this optimization removed approximately 900 latch drivers.

Several smaller utilities were written to check beta ratios, size power rails, and determine power dissipation. Under the category of future work, several ideas are also presented concerning an automated approach for performing timing analysis and logic optimization. The methodology discussed consists of the following steps: 1) levelization in order to identify paths which exceed the targeted gate depth, 2) optimal logic synthesis for logic which has less then 10 inputs/outputs (the majority of logic within a design is generated by a relatively small number of inputs and produces a small number of outputs), 3) logic retiming across latch boundaries, and 4) close coupling for the steps of parasitic extraction, delay calculation, static timing analysis, and buffer sizing/selection. None of these steps are innovative, but the overall automated system can significantly improve design time and reduce tedious operations that would otherwise be performed by hand.

The dissertation concludes with a brief discussion of functional verification, especially as it pertains to random testing. This approach focused on three areas: computation units, chip-level, and system level. Several different simulation environments were created to support generation and self-checking validation of both random numerical operations and random instruction sequences. Over 5 million computations were simulated for each functional unit and over 200 million instructions were simulated in all.

Appendix A
Aurora III Chip Layout



Appendix B
Corrections to Add Unit Logic

1 Generation of Guard, Round, and Sticky Bits

The nomenclature used for the following equations is defined in [Quach91a], [Quach91c]. Primary inputs are defined as:

expAbs exponent of larger operand

G intermediate guard bit that is generated from alignment shifter

R intermediate round bit that is generated from alignment shifter

S intermediate sticky bit that is generated from alignment shifter

The final guard, round, and sticky bits are given the names, respectively: ***bn***, ***bnp1***, ***s***. This logic is necessary because the alignment shifter does not zero out the guard and/or round bits for a shift greater than the width of the mantissa (53-bits). As a result of the hidden bit, the sticky bit may need to be set for a shift greater than the mantissa width; this action is not performed by the alignment logic that generates the intermediate sticky bit. The revised equations, which differ from the those presented in the above papers, are:

Es54 = expAbs[5] & expAbs[4] & ~expAbs[3] & expAbs[2] & expAbs[1] & ~expAbs[0]

Esgt55A = expAbs[5] & expAbs[4] & ~expAbs[3] & expAbs[2] & expAbs[1] & expAbs[0]

Esgt55B = NAND{~expAbs[10:6]}

Esgt55C = expAbs[5] & expAbs[4] & expAbs[3]

Esgt55 = Esgt55A + Esgt55B + Esgt55C

Es09 = AND{expAbs[10:1]}

bn = (~Es09 + Es10) & ~Es54 & ~Esgt55 & G

bnp1 = Es54 + (~Es54 & (Es09 + Esgt55 + ~R))

s = (~Es09 & ~Es54 & ~S) + (Es54 & R) + (Es54 & S) + Esgt55

2 Additional Rounding Logic

The nomenclature used for the following equations is defined in [Quach91a], [Quach91c]. This section also summarizes revised versions for some of the logic presented in these papers. Primary inputs are defined as:

addORsub operation (addition = 0, subtraction = 1)

Eo effective operation, considering sign of operands (addition = 0, subtraction 1)

Ccabar carry-out of mantissa adder (Cin = 0 carry tree)

Es09 AND of the complement of the high bits of larger exponent

Es10bar lsb of larger exponent

Cexp carry-out of exponent adder, used to determine which operand is larger

Fs00 msb of A+B+1 sum of mantissa adder

Fs10 msb of A+B+1 sum of mantissa adder

S51 XOR of lsb's of input operands

S52 XOR of lsb+1's of input operands

bn guard bit

bnp1 round bit

s sticky bit

Sa sign of RS operand

Sb sign of RT operand

RM rounding mode (RN = 0, RZ = 1, RP = 2, RM = 3)

Normalization of the result falls into several classes: one right shift (***ORS***), no shift (***NXS***), many left shift (***MLS***), one left shift (***OLS***). The conditions which define these classes are:

ORS = ~Eo & ~Ccabar;

NXS = (~Eo & Ccabar) + ((Eo & ~Es09) & ((Fs10 & ~(bn + bnp1 + s)) + (Fs00 & (bn + bnp1 + s))))

MLS = Eo & Es09

OLS = (Eo & ~Es09) & ((~Fs10 & ~(bn + bnp1 + s)) + (~Fs10 & (bn + bnp1 + s)))

The result sign is defined as:

Se = (Cexp & Sa) + (~Cexp & ~Ccabar & ~Sb & addORsub) + (~Cexp & ~Ccabar & Sb & ~addORsub) +(~Cexp & Ccabar & Sa)

For each of the rounding modes, two results are produced: ***CinRX***, which indicates whether a round is needed, and ***qRX*** which is the least-significant-bit to be shifted in during a normalization.

Round-to-nearest:

A = Eo & Es09 & Es10bar & (~bn + Fs00) & (S52 + ~bn)

B = (~Eo & S52 & (bn + bnp1 + s + S51)) + (Eo & Es09 & ~Es10bar);

C = ~Eo & bn & (bnp1 + s + S52);

D = (Eo & ~Es09 & (~bn + (bn & ~bnp1 & ~s & S52))) + (Eo & Es09 & ~Es10bar & bn & S52);

E = Eo & ~Es09 & ~bn & (~bnp1 + ~s);

CinRN = A + (~Ccabar & B) + (Ccabar & C) + (Fs00 & D) + (~Fs00 & E)

qRN = (~bn & bnp1 & s) + (bn & ~bnp1)

Round-to-Zero:

CinRZ = (~Es09 & Eo & ~(bn + bnp1 + s)) + (Es09 & Eo & ~bn & ~Ccabar);

qRZ = bn ^ (s + bnp1);

Round-to-plus-infinity:

roundMLSenCin = ~bn + (~Es10bar + Fs00) & (~Es10bar + Ccabar)

CinRPpNXS = ((((Eo & (~Se + (~bn & ~bnp1 & ~s)))) + (~Eo & (~Se & (bn + bnp1 + s)))));

CinRPppNXS = NXS & S52 & (CinRPpNXS ^ Ccabar)

CinRPpOLS = (((~Se & ~bn) + (Se & ~bn & ~bnp1 & ~s)))

CinRPppOLS = OLS & CinRPpOLS & S52

CinRPpMLS = MLS & roundMLSenCin & ((~Se & bn & S52 & ~Ccabar) + (Eo & ~bn & S52 & ~Ccabar))

CinRP = (ORS & ~Se & (S52 + bn + bnp1 + s)) + CinRPppNXS + CinRPppOLS + CinRPpMLS

qRP = (~Se & bn) + (Se & ((~bn & s) + (~bn & bnp1) + (bn & ~bnp1 & ~s))) + (MLS & bn)

Round-to-minus-infinity:

CinRMpNXS = ((((Eo & (Se + (~bn & ~bnp1 & ~s)))) + (~Eo & (Se & (bn + bnp1 + s)))));

CinRMppNXS = NXS & S52 & (CinRMpNXS ^ Ccabar)

CinRMpOLS = (((Se & ~bn) + (~Se & ~bn & ~bnp1 & ~s)))

CinRMppOLS = OLS & CinRMpOLS & S52

CinRMpMLS = MLS & roundMLSenCin & ((Se & Eo & S52 & ~Ccabar) + (Eo & ~bn & S52 & ~Ccabar))

CinRM = (ORS & Se & (S52 + bn + bnp1 + s)) + CinRMppNXS + CinRMppOLS + CinRMpMLS

qRM = (Se & bn) + (~Se & ((~bn & s) + (~bn & bnp1) + (bn & ~bnp1 & ~s))) + (MLS & bn)

Finally, a signal is needed to determine whether the output of the mantissa adder should be complemented:

ComplS0RPpNXS = ~S52 & ~(CinRPpNXS ^ Ccabar)

ComplS0RMpNXS = ~S52 & ~(CinRMpNXS ^ Ccabar)

roundMLSenComplS0 = (bn & Es10bar & ~S52 & Fs00) + (bn & Es10bar & ~S52 & Ccabar)

ComplS0pMLS = MLS & ((~Eo & ~bn & ~RM[0] & ~S52) + (Se & bn & ~RM[0] & ~S52) + (~RM[0] & ~S52 & Ccabar) + (RM[0] & ~S52 & Ccabar) + (~Se & bn & RM[0] & ~S52) + (~Se & ~Eo & RM[0] & ~S52) + roundMLSenComplS0)

ComplS0 = (RM[0] & OLS & ~(CinRMpOLS + S52)) + (~RM[0] & OLS & ~(CinRPpOLS + S52)) + (~RM[0] & ComplS0RPpNXS & NXS) + (RM[0] & ComplS0RMpNXS & NXS) + ComplS0pMLS

Appendix C

References used for plots of clock frequency vs. year and transistor count vs. year

[Benschneider89] [Benschneider89] [Birman90] [Darley90] [Elkind87] [Fossum85] [Fuccio88] [Gavrielov86] [Gosling81] [Ho85] [Ide92] [Jouppi88] [Jouppi89] [Jouppi89] [Kaneko89] [Kasai85] [Kohn89] [Komal85] [Komori89] [Kawakami86] [Kawasaki89] [Lu88] [McAllister86] [Molnar89] [Montoye90] [Nakayama89] [Okamoto91] [Oehler90] [Papamichalis88] [Rowen88] [Schutz91] [Shimazu89] [Sit89] [Sohie88] [Staver87] [Steiss91] [Takeda85] [Takla84] [Taylor90] [Tran85] [Troutman86] [Ware82] [Ware84] [Wolrich84]

Bibliography

[Atkins68] Atkins, D., "Higher-Radix Division Using Estimates of the Divisor and Partial Remainders," *IEEE Transactions on Computers*, C-17:925-934, 1968.

[Benschneider89] Benschneider, B. J., et al., "A 50MHz Uniformly Pipelined 64b Floating-Point Arithmetic Processor," *ISSCC*, 1989.

[Benschneider89] Benschneider, B. J., et al., "A Pipelined 50-MHz CMOS 64-bit Floating Point Arithmetic Processor," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 5, pp. 1317-1325, Oct. 1989.

[Bewick88] Bewick, G., et al., "Approaching a Nanosecond: A 32 bit Adder," *IEEE Press*, pp. 221-226, 1988.

[Birman90] Birman, M., et al., "Developing the WTL3170/3171 Sparc Floating-Point Corprocessors," *IEEE Micro*, pp. 55-64, Feb. 1990.

[Bose87] Bose, B. K., et al., "Fast Multiply and Divide for a VLSI Floating-Point Unit," *Proceedings of Eighth Symposium on Computer Arithmetic*, pp. 87-94, 1987.

[Brown92a] Brown, R.B., et al., "GaAs RISC Processors," Invited Paper, *1992 GaAs IC Symposium*, pp. 81-84, Oct. 1992.

[Brown92b] Brown, R. B., "Compound Semiconductor Device Requirements for VLSI," *Gallium Arsenide and Related Compounds 1992, pp. 857-862,* Sep. 1992.

[Brown93] Brown, R. B., et al., "Gallium Arsenide Process Evaluation Based on a RISC Mircroprocessor Example," *IEEE Journal of Solid-State Circuits Conference*, vol. 28, no. 10, pp. 1030-1037, Sep. 1993.

[Chandna94] Chandna, A., "GaAs MESFET Static RAM Design For Embedded Applications," Doctoral Thesis, University of Michigan, 1994.

[Chen94] Chen, T., et al., "A Performance Study of Software and Hardware Data Prefetching Schemes," *International Symposium on Computer*

Architecture, Chicago, Illinois, pp. 223-232, May 1994.

[Cvetanovic94] Cvetanovic, Z., "Chacterization of Alpha AXP Performance Using TP and SPEC Workloads," *International Symposium on Computer Architecture*, Chicago, Illinois, pp. 60-70, May 1994.

[Darley90] Darley, M., et al., "The TMS390C602A Floating-Point Coprocessor for Sparc Systems," *IEEE Micro*, pp. 36-47, Jun. 1990.

[Diefendorff94] Diefendorff, R., et al., "Evolution of the PowerPC Architecture," *IEEE MICRO*, pp. 34-49, April 1994.

[Dobberpuhl92] Dobberpuhl, D., "A 200-MHz 64-b Dual-Issue CMOS Microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 11, Nov. 1992.

[Elkind87] Elkind et al., "A Sub 10ns Bipolar 64 Bit Integer/Floating Point Processor Implemented on Two Circuits," *IEEE Press*, pp. 101-104, 1987.

[Ercegovac87] Ercegovac, M. D., et al., "On-the-Fly Conversion of Redundant into Conventional Represenations," *IEEE Transactions on Computing*, vol. C-36, pp. 895-897, Jul. 1987.

[Ercegovac89] Ercegovac, M. D., et al., "On-the-Fly Rounding for Division and Square Root," *Proc. 9th IEEE Symposium on Computer Arithmetic*, Santa Monica, CA, pp. 169-173, Sept. 1989.

[Ercegovac97] Ercegovac, M. D., et al., "On-the-Fly Rounding," *IEEE Transactions on Computers*, vol. 41, No. 12, pp. 1497-1503, Dec. 1992.

[Fandrianto87] Fandrianto, J. "Algorithm for High Speed Shared Radix 4 Division and Radix 4 Square-Root," *Proc. 8th IEEE Symposium on Computer Arithmetic*, Como, Italy, pp. 73-79, May 1987.

[Fandrianto89] Fandrianto, J. "Algorithm for High Speed Shared Radix 8 Division and Radix 8 Square-Root," *Proc. 9th IEEE Symposium on Computer Arithmetic*, Santa Monica, CA, pp. 68-75, Sep. 1989.

[Fossum85] Fossum, Grundmann, Blaha, "Floating Point Processor for the VAX 8600," *IEEE Press*, 1985.

[Fu92] Fu, J. W. C., et al., "Stride directed prefetching in scalar processors," In *Proc. of the 25th International Symposium on Microarchitecture*, pp.102-110, Dec. 1992.

[Fuccio88] Fuccio, M. L.,, et al., "The DSP32C: AT&T's Second-Generation

Floating-Point Digital Signal Processor," *IEEE Micro*, pp. 30-48, Dec. 1988.

[Fulkerson91] Fulkerson, D. E., "Feedback FET Logic: A Robust, High-Speed, Low-Power GaAs Logic Family," *IEEE Journal of Solid-State Circuits*, Vol. 26, pp. 70-74, Jan. 1991.

[Gavrielov86] Gavrielov, M., et al., "The NS32081 Floating-Point Unit: Architecture and Implementation," *IEEE Micro*, pp. 6-12, Apr. 1986.

[Gosling81] Gosling, et al., "A Chip-Set for a High-Speed Low-Cost Floating-Point Unit," *IEEE Press*, 1981.

[Hallmark94] Hallmark, J., et al., "0.9V DSP Blocks: A 15ns SRAM and a 45ns 16-bit Mutiply/Accumulator," *1994 GaAs IC Symposium*, Philadelphia, Pennsylvania, pp. 55-58, Oct. 1994.

[Harata87] Harata, Y., et al., "A High-Speed Multiplier Using a Redundant Binary Adder Tree," *IEEE Journal of Solid-State Circuits*, vol. SC-22, no. 1, pp. 28-34, Feb. 1987.

[Ho85] Ho, et al., "A High Performance 1.25 microns CMOS Floating Point Multiply/Accumulate Chip," *IEEE Press*, 1985.

[Hokenek90] Hokenek, E., et al., "Leading-zero Anticipator (LZA) in the IBM RISC System/6000 Floating-Point Execution Unit," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 71-77, Jan. 1990.

[Iacobovici88] Iacobovici, S., "A Pipelined Interface for High Floating-Point Performance with Precise Exceptions," *IEEE Micro*, pp. 77-87, Jun. 1988.

[Ide92] Ide, N., et al., "A 320 MFLOPS CMOS Floating-Point Processing Unit For Superscalar Processors," *IEEE Custom Integrated Circuits Conference*, pp. 30.2.1-30.2.4, 1992.

[IEEE88] *An American National Standard: IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard No. 754, American National Standards Institute, Washington, DC, 1988.

[Johnson91] Johnson, M., *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Jouppi88] Jouppi, N. "MultiTitan Floating Point Coprocessor," *DEC Internal Report*, 1988.

[Jouppi89] Jouppi, N., et al., "A Unified Vector/Scalar Floating-Point

Architecture," *Association for Computing Machinery*, 1989.

[Jouppi89] Jouppi, N., et al., "A Unified Vector/Scalar Floating-Point Architecture," *Association for Computing Machinery*, pp. 134-143, 1989.

[Jouppi90] Jouppi, N., "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 364-373, May 1990.

[Kaneko89] Kaneko, et al., "A VLSI RISC with 20-MFLOPS Peak, 64-bit Floating-Point Unit," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 5, pp. 1331-1340, Oct. 1989.

[Kasai85] Kasai, M., et al., "A Single Chip CMOS Floating Point Signal Processor," *CICC*, 1985.

[Kawakami86] Kawakami, Y., et al., "A 32b Floating Point CMOS Digital Signal Processor," *ISSCC*, pp. 86-7, 1986.

[Kawasaki89] Kawasaki, S., et al., "A Floating-Point VLSI Chip for the TRON Architecture: An Architecture for Reliable Numerical Programming," *IEEE Micro*, pp. 26-44, Jun. 1989.

[Kayssi93] Kayssi, A., et al., "Delay Modeling for GaAs DCFL Circuits," *GaAs IC Symposium*, San Jose, California, pp. 67-70, Oct. 1993.

[Kayssi93] Kayssi, A., et al., "The impact of signal transition time on path delay computation," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 40, no. 5, pp. 302-309, May 1993.

[Kohn89] Kohn, et al., "A 1,000,000 Transistor Microprocessor," *ISSCC*, 1989.

[Komal85] Komal, A., et al., "An IEEE Standard Floating Point Chip," *ISSCC*, pp. 18-19, 1985.

[Komori89] Komori, S., et al., "A 40-MFLOPS 32-bit Floating-Point Processor with Elastic Pipeline Scheme," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 5, pp. 1341-1347, Oct. 1989.

[Klaiber91] "An architecture for software-controlled data prefetching," In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 43-53, 1991.

[Laha88] Laha, S., et al., "Accurate Low-Cost Methods for Performance

Evaluation of Cache Memory Systems," *IEEE Transactions on Computers*, vol. 37, no. 11, pp. 1325-1336, 1988.

[Lehman61] Lehman, M, et al., "Skip techniques for high-speed carry propagation in binary arithmetic units," *IRE Transaction on Electronic Computers*, Dec. 1961.

[Ling81] Ling, H., "High-Speed Binary Adder," *IBM Journal of Research and Development*, vol. 25, no. 3, pp. 156-166, May 1981.

[Liu93] Liu, L., et al., "Cache Sampling by Sets," *IEEE Transactions on VLSI Systems*, Vol. 1, No. 2, pp. 98-105, 1993.

[Lu88] Lu, P. et al., "A 30-MFLOP 32b CMOS Floating-Point Processor," *ISSCC*, pp. 28-29, 1988.

[Magenheimer87] Magenheimer, D. J., et al., "Integer Multiplication and Division on the HP Precision Architecture," *Association for Computing Machinery*, pp. 90-99, 1987.

[Majerski67] Majerski, S., "On determination of optimal distribution of carry skips in adders," *IEEE Transactions on Computers*, EC-16, Feb. 1967.

[Majerski85] Majerski, S., "Square-Rooting Algorithms for High-Speed Digital Circuits," *IEEE Transactions on Computers*, Vol. C-34, No. 8, Aug. 1985.

[Makino93] Makino, H., et al., "A 8.8-ns 54x54-bit Multiplier Using New Redundant Binary Architecture," *International Conference on Computer Design*, pp. 202-205, 1993.

[McAllister86] McAllister, W., et al., "An NMOS 64b Floating-Point Chip Set," *ISSCC*, pp. 34-35, 1986.

[Molnar89] Molnar, et al., " A 40MHz 64-bit Floating-Point Co-Processor," *IEEE Solid-State Circuits Conference*, 1989.

[Montoye90] Montoye, R. K., et al., "Design of the IBM RISC System/6000 Floating Point Execution Unit," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 59-70, Jan. 1990.

[Montoye90] Montoye, et al., "An 18ns 56-bit Multiply-Adder Circuit," *ISSCC*, 1990.

[Montuschi93] Montuschi, P. et al., "Reducing Iteration Time When Result Digit is Zero for Radix 2 SRT Division and Square Root with Redundant Remainders," *IEEE Transactions on Computers*, vol. 42, no. 2, pp.

239-246, Feb. 1993.

[Mulder91] Mulder, J., "An area model for on-chip memories and its application," *Journal of Solid-State Circuits*, vol. 26, no. 2, pp. 98-106, 1991.

[Nagle90] Nagle, D., "Floating Point Simulation for the GaAs Micro-Supercomputer," University of Michigan - Internal Report, 1990.

[Nagle91] Nagle, D., "Hiding Latency in the GaAs Floating Point Unit," University of Michigan - Internal Report, Apr. 1991.

[Nakayama89] Nakayama, T., et al., "A 6.7-MFLOPS Floating-Point Coprocessor with Vector/Matrix Instructions," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 5, pp. 1324-1330, Oct. 1989.

[Oehler90] Oehler, R. R., et al., "IBM RISC System/6000 Processor Architecture," *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 23-36, Jan. 1990.

[Oettel92] Oettel, R., Internal Report, Cascade Design Automation, 1992.

[Okamoto91] Okamoto, F., et al., "A 200MFLOPS 100MHz 64b BiCMOS Vector-Pipelined- Processor," *ISSCC*, pp. 256-7, 1991.

[Oklobdzija85] Oklobdzija, V. G., et al., "Some Optimal Schemes for ALU Implementation in VLSI Technology," *Proceedings of Seventh Symposium on Computer Arithmetic*, pp. 2-8, 1985.

[Papamichalis88] Papamichalis, P., et al., "The TMS320C30 Floating-Point Digital Signal Processor," *IEEE Micro*, pp. 13-29, Dec. 1988.

[Peng87] Peng, V., et al., "On the Implementation of Shifters, Multipliers, and Dividers in VLSI Floating-Point Units," *IEEE Press*, pp. 95-101, 1987.

[Poursepanj94] Poursepanj, A., et al., "The PowerPC 603 Microprocessor: Performance Analysis and Design Trade-offs," *IEEE Press*, 1063-6390/94, 1994.

[Putti93] Putti, D., "The Design and Implementation of the Aurora III Divide Unit," Technical Report, University of Michigan, 1993.

[Quach90] Quach, N., et al., "An Improved Algorithm For High-Speed Floating-Point Addition," Computer Systems Laboratory, Stanford, 1990.

[Quach90] Quach, N., et al., "High-Speed Addition in CMOS," Stanford

Technical Report: CSL-TR-90-415, Feb. 1990.

[Quach91a] Quach, N., et al., "Design and Implementation of the SNAP Floating-Point Adder," Stanford Technical Report: CSL-TR-91-501, Dec. 1991.

[Quach91b] Quach, N., et al., "Leading One Prediction - Implementation, Generalization, and Application," Stanford Technical Report: CSL-TR-91-463, Mar. 1991.

[Quach91c] Quach, N., et al., "On Fast IEEE Rounding," Stanford Technical Report: CSL-TR-91-459, Jan. 1991.

[Quach91d] Quach, N., et al., "Suggestions for Implementing a Fast IEEE Multiply-Add-Fused Instruction," Stanford Technical Report, CSL-TR-91-483, Jul. 1991.

[Riepe93] Riepe, M., "A 53-bit RBSD Parallel Array Multiplier with IEEE Double Precision Floating Point Rounding Implemented in 0.6um GaAs DCFL," Technical Report, University of Michigan, 1993.

[Riepe94] Riepe, M., et al., "Implementing IEEE Rounding in Parallel-Array Floating-Point Multipliers," submitted to *12th IEEE Symposium on Computer Arithmetic*, 1994.

[Robertson58] Robertson, J., "A New Class of Digital Division Methods," *IRE Transactions on Electronic Computing*, EC-7, pp. 218-222, 1958.

[Rowen88] Rowen, C., et al., "The MIPS R3010 Floating-Point Coprocessor," *IEEE Micro*, pp. 53-62, Jun. 1988.

[Schutz91] Schutz, J., "A CMOS 100MHz Microprocessor," *ISSCC*, pp. 90-91, 1991.

[Shimazu89] Shimazu, Y., et al., "A 50MHz 24b Floating-Point DSP," *ISSCC*, pp. 44-45, 1989.

[Sit89] Sit, H. P., et al., "An 80 MFLOPS Floating-Point Engine in the Intel i860 Processor," *IEEE Press*, pp. 374-379, 1989.

[Smith85] Smith, J. E., et al., "Varieties of Decoupled Access/Execute Computer Architectures," Internal Report, Unversity of Wisconsin-Madison, pp. 577-586, 1985.

[Smith86] Smith, J. E., et al., "A Simulation Study of Decoupled Architecture Computers," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 692-702, Aug. 1986.

[Smith87] Smith, J. E., et al., "The ZS-1 Central Processor," *Association for Computing Machinery*, pp. 199-204, 1987.

[Smith88] Smith, J., et al., "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers*, vol. 37, no. 5, pp. 562-573, May 1988.

[Smith91] Smith, M., "Tracing With Pixie," Center for Integrated Systems, Stanford University, Apr. 1991.

[Smith92] Smith, M., "Support for Speculative Execution in High-Performance Processors," PhD dissertation, Stanford University, Nov. 1992.

[Sohie88] Sohie, G., et al., "A Digital Signal Processor with IEEE Floating-Point Arithmetic," *IEEE Micro*, pp. 49-67, Dec. 1988.

[Staver87] Staver, D., "A 30-MFLOPS CMOS Single Precision Floating Point Multiply/Accumulate Chip," *ISSCC*, pp. 274-275, 1987.

[Steiss91] Steiss, et al, "A 65MHz Floating-Point Coprocessor for a RISC Processor," *ISSCC*, 1991.

[Stritter90] Stritter, S., et al., "Preliminary Benchmark of Vitesse GaAs," Internal Report, MIPS Corp., Feb. 1990.

[Takeda85] Takeda, K., et al., "A Single-Chip 80-bit Floating Point Processor," *IEEE Journal of Solid-State Circuits*, vol. SC-20, no. 5, pp. 986-991, Oct. 1985.

[Takla84] Takla, N., et al., "A Monolithic 64 Bit Floating-Point Coprocessor," *IEEE Journal of Solid-State Circuits*, vol SC-19, no. 4, pp. 538-539, Aug. 1984.

[Taylor90] Taylor, G., et al., "A 100 MHz Floating Point/Integer Processor," *IEEE Integrated Circuits Conference*, pp. 24.5.1-24.5.4, 1990.

[Tran85] Tran, T., et al., "A 1.0-Micron CMOS 32-Bit IEEE Format Floating Point Chip Set for Digital Signal Processing," *IEEE Custom Integrated Circuits Conference*, pp. 281-284, 1985.

[Troutman86] Troutman, W., et al., "Design of a Standard Floating-Point Chip," *IEEE Journal of Solid-State Circuits*, vol. SC-21, no. 3, pp. 396-399, Jun. 1986.

[Turrini89] Turrini, S., "Optimal group distribution in carry-skip adders," Internal Technical Report, Digital Equipment Corp., Palo Alto, California, 1989.

[Unger77] Unger, S. H., et al., "Tree Realizations of Iterative Circuits," *IEEE Transactions on Computers*, vol. C-26, no. 4, pp. 365-383, Apr. 1977.

[Upton91] Upton, M., "The Design of an Optimal Price-Performance GaAs Floating Point Chip," University of Michigan - Internal Report, Apr. 1991.

[Upton94] Upton, M., et al., "Resource Allocation in a High Clock Rate Microprocessor," In the *21st Annual International Symposium on Computer Architecture*, Chicago, Illinois, pp. 98-109, Oct. 1994.

[Uya84] Uya, et al., "A CMOS Floating Point Multiplier," *IEEE Journal of Solid-State Circuits*, vol. SC-19, no. 5, October 1984.

[Vuillemin83] Vuillemin, J. "A Very Fast Multiplication Algorithm for VLSI Implementation," INTEGRATION: *The VLSI Journal*, pp. 39-52, 1983.

[Ware82] Ware, F. A., et al., "64 Bit Monolithic Floating Point Processors," *IEEE Journal of Solid-State Circuits*, vol. SC-17, no. 5, pp. 898-906, Oct. 1982.

[Ware84] Ware, "Pipelined IEEE Floating Point Processors," *IEEE Press*, 1984.

[Williams86] Williams, T., et al., "SRT Division Diagrams and Their Usage in Designing Custom Integrated Circuits for Division," Stanford Technical Report, CSL-TR-87-326, Nov. 1986.

[Williams91] Williams, T., et al., "A Zero-Overhead Self-Timed 160ns 54b CMOS Divider," *ISSCC*, pp. 98-99, 1991.

[Wilson61] Wilson, J. B., "An Algorithm for Rapid Binary Division," IRE *Transactions on Electronic Computing*, EC-10, pp. 662-670, 1961.

[Wolfe92] Wolfe, A., "The Split Data Cache Model: A Cache Optimization for Superscalar RISC Microprocessors," Technical Report, Princeton University, 1992.

[Wolrich84] Wolrich, G., et al., "A High Performance Floating Point Coprocessor," *IEEE Journal of Solid-State Circuits*, vol. SC-19, no. 5, pp. 690-696, Oct. 1984.

[Wong92] Wong, D., "Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations," *IEEE Transactions on Computers*, vol. 41, no. 8, pp. 981-995, Aug. 1992.

[Yeh93] Yeh, T., "Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanism Designs for High Performance Superscalar Processors," Doctoral Thesis, University of Michigan, 1993.